Programmieren von LegoMindstorms-Robotern mit NQC

(Version 3.03, Oct 2, 1999)

by Mark Overmars

Department of Computer Science
Utrecht University
P.O. Box 80.089, 3508 TB Utrecht
the Netherlands

Deutsche Übersetzung von
Martin Breuner
Gewerbliche Schule Bad Mergentheim
Seegartenstrasse 16
97980 Bad Mergentheim
Deutschland

Im Dezember 1999
Die aktuelle Version von NQC ist 2.0, die des Command Center 3.03

Einleitung

Die Roboter **Lego- MindStorms** und **CyberMaster** sind wundervolle neue Spielzeuge, aus denen die tollsten Roboter gebaut werden können. Diese können so programmiert werden, dass sie die verschiedensten schwierigen Aufgaben erfüllen können.
Die original Lego- Software ist sehr leicht zu handhaben. Leider ist sie in der Funktionalität ziemlich stark eingeschränkt. Folglich kann sie nur für einfache Aufgaben verwendet werden. Um alle Möglichkeiten des RCX ausschöpfen zu können, benötigt man eine leistungsfähigere Programmiersprache.
NQC ist eine solche Programmiersprache, die von Dave Baum besonders für die Lego-Roboter geschrieben wurde.
Wenn du nie zuvor ein Programm geschrieben hast, keine Angst. NQC ist wirklich sehr einfach zu erlernen und dieser Leitfaden erklärt alles was du wissen mußt, um deinen Roboter mit NQC zu programmieren.
Da die Lego-Roboter mit NQC so leicht zu programmieren sind, bietet dieses die Möglichkeit auf spielerische Weise das Programmieren zu erlernen.
Damit das Schreiben der Programme noch einfacher wird, gibt es das **RCX-Command Center**. Dieses Hilfsprogramm hilft dir deine Programme zu schreiben, auf den Roboter zu übertragen und sogar die direkte Kontrolle vom Computer aus über den Roboter zu übernehmen.
Das **RCX-Command Center** kann kostenlos über folgende Internet-Adresse bezogen werden.

http://www.cs.uu.nl/people/markov/lego/

RCX-Command Center läuft auf Windows PC's (' 95, '98, NT). Beachte bitte, dass, die original Lego- Software auf dem Rechner zumindest einmal gestartet werden mußte, damit das **RCX-Command** Center verwendet werden kann. (Die Lego-Software installiert bestimmte Bestandteile, die das **RCX-Command Center** benutzt).
Falls du die Original Lego-Software auf deinem Rechner nicht installieren kannst oder willst, so kannst du die notwendige Einstellung auch auf folgende Weise erlangen:

- Lege die Original Lego-Software CD in Dein CD-Laufwerk ein.
- Gib in die Befehlszeile bei ->Start ->Ausführen den Befehl ein: **d:\REGSVR32.EXE d:\system\spirit.ocx**
  wobei **d** der Laufwerksbuchstabe deines CD-Laufwerkes ist.

Für weiter Informationen lies bitte in der Datei RcxCC.doc nach.

Die Sprache NQC kann auch auf anderen Betriebssystemen verwendet werden. Du kannst sie vom Web von folgender Adresse downloaden.

http://www.enteract.com/~dbaum/lego/nqc/

Die meisten Inhalte dieses Leitfadens wenden sich auch an die Anwender anderer Betriebssysteme (vorausgesetzt, du verwendest die NQC-Version 2,0 oder höher), allerdings mit Einschränkungen bei manchen Funktionen und bei der Farbcodierung der Befehle.
In diesem Leitfaden nehme ich an, daß du den MindStorms- Roboter hast. Die meisten Inhalte treffen auch auf den CyberMaster zu, obgleich auch hier nicht alle Funktionen für den Cybermaster vorhanden sind. Auch die Namen z.B. der Motoren sind unterschiedlich. Also mußt du die Beispiele ein wenig ändern, damit dein Cybermaster- Roboter die Befehle ausführen kann.
Schließlich möchte ich Dave Baum für das Entwickeln von NQC danken. Ebenso vielen Dank an Kevin Saddi, der die erste Hälfte der ersten Version dieses Leitfadens erstellt hat.

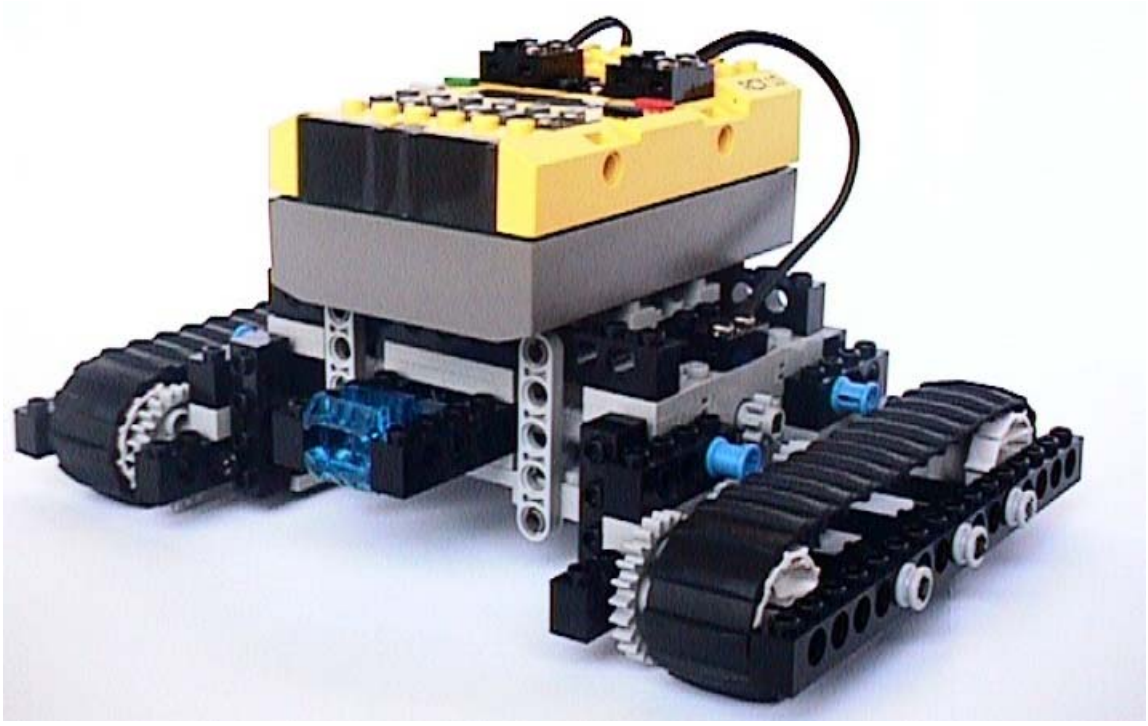
Inhalt

Einleitung....2
Inhalt....3
2.Dein erstes Programm in NQC....5
2.1.Starten von RCX-Command Center....5
2.2.Wir schreiben unser erstes Programm....6
2.3.Wir lassen das Programm ablaufen....7
2.4.Fehler im Programm ??....7
2.5.Ändern der Geschwindigkeit....8
2.6.Zusammenfassung....8
3.Ein interessanteres Programm....8
3.1.Der Roboter dreht sich....8
3.2.Befehle wiederholen....9
3.3.Kommentare hinzu fügen....10
3.4.Zusammenfassung....10
4.Wir verwenden Variablen....11
4.1.Unser Roboter fährt eine Spirale....11
4.2.Zufallszahlen....12
4.3.Zusammenfassung....12
5.Steuerbefehle....13
5.1.Die if - Anweisung....13
5.2.Die do- Anweisung....14
5.3.Zusammenfassung....14
6.Sensoren....15
6.1.Abfragen eines Sensors....15
6.2.Reagieren auf einen Berührungssensor....16
6.3.Lichtsensoren....16
6.4.Zusammenfassung....17
7.Tasks (Aufgaben) und Unterprogramme....18
7.1.Tasks....18
7.2.Unterprogramme....18
7.3.Inline- Funktionen....19
7.4.Die Definition von Makros....20
7.5.Zusammenfassung....21
8.Der RCX macht Musik....22
8.1.Vorprogrammierte Klänge....22
8.2.Musik....22
8.3.Zusammenfassung....23
9.Mehr über Motoren....24
9.1.Motoren auslaufen lassen....24

9.2.Weiterentwickelte Befehle....24
9.3.Unterschiedliche Geschwindigkeiten....25
9.4.Zusammenfassung....25
10.Mehr über Sensoren....26
10.1.Sensor- Modus und Sensor- Art....26
SENSOR_MODE_RAW....26
SENSOR_MODE_BOOL....26
SENSOR_MODE_PERCENT....26
SENSOR_MODE_EDGE und SENSOR_MODE_PULSE....26
ClearSensor()....26
10.2.Der Umdrehungs- Sensor....27
10.3.Mehrere Sensoren an einem Eingang....27
10.4.Ein Annäherungssensor....28
10.5.Zusammenfassung....29
11.Parallele Tasks (Aufgaben)....30
11.1.Stoppen und erneutes Starten von Tasks....31
11.2.Standardtechnik der Semaphore....31
11.3.Zusammenfassung....32
12.Die Roboter unterhalten sich....33
12.1.Ordnungen....33
12.2. Wählen eines Anführers....34
12.3.Vorsichtsmaßnahmen....34
Achte darauf, dass beim Übertragen eines Programms nur ein RCX eingeschaltet ist!....34
Überlege dir ein Protokoll, das Übertragungsfehler erkennt!....35
12.4.Der RCX im Flüstermodus....35
12.5. Zusammenfassung....35
13.Noch mehr Befehle....36
13.1.Timer....36
13.2.Das Display des RCX....36
13.3.Speichern von Informationen....38
14.NQC- Befehle im Überblick....39
Statements....39
Bedingungen....40
Ausdrücke....41
Mathematische Operationen....42
RCX Funktionen....43
RCX Konstanten....44
Schlüsselworte....44
15. Abschließende Bemerkungen....45
Anmerkungen des Übersetzers....45

2. Dein erstes Programm in NQC

In diesem Kapitel zeige ich dir, wie man ein extrem einfaches Programm schreibt.
Wir werden einen Roboter programmieren, der für 4 Sekunden vorwärts, danach für weitere 4 Sekunden rückwärts fährt und dann stoppt. Nicht sehr spektakulär, aber es dient dazu, dich in die Programmierung mit NQC einzuführen, und es zeigt dir, wie einfach es ist.
Aber bevor wir ein Programm schreiben können, benötigen wir zuerst einen Roboter.
Baue den in der **CONSTRUCTOPEDIA** (das ist das größere Buch, das bei deinem MindStorms –Baukasten dabei ist) auf den Seiten 12 bis 16 und 21 bis 25 beschriebenen **Roverbot**. Achte dabei besonders auf den Anschluß der Kabel. Dieses ist wichtig, damit dein Roboter in die korrekte Richtung fährt. Dein Roboter sollte etwa wie dieser aussehen:

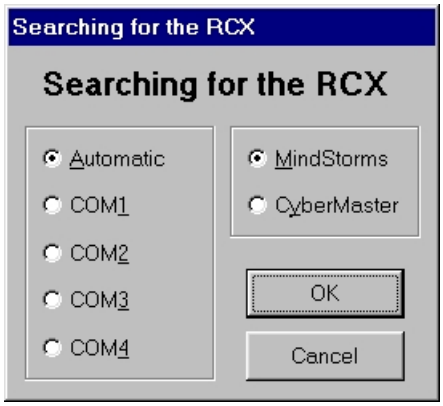


Überprüfe auch, ob die Infrarot- Sendestation richtig an deinen Computer angeschlossen ist, und dass sie auf große Übertragungsstrecke eingestellt ist. Wenn du nicht weißt wie das geht, siehe in deinem **Mindstorms User Guide** auf den Seiten 14 und 15 nach. (Das ist das kleinere Buch, das bei deinem MindStorms –Baukasten dabei ist.)

2.1. Starten von RCX-Command Center

Wir schreiben unsere Programme mit dem **RCX-Command Center.**
Starte das **RCX-Command Center,** indem du auf das Symbol RcxCC doppelklickst.
(Oder **Start** -> **Programme** -> **Rcx Command Center**)

Ich nehme an, daß du bereits **RCX-Command Center** installiert hast. Wenn nicht, downloade es vom Internet, (Adresse: http://www.cs.uu.nl/people/markov/lego/) und installiere es in ein Verzeichnis deiner Wahl.

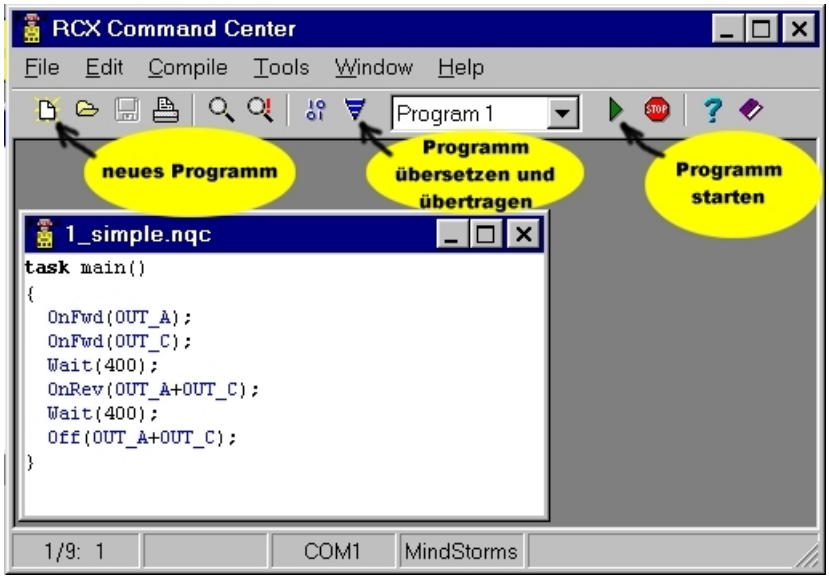


Das Programm fragt dich nach dem RCX. Schalte ihn ein und stelle ihn mit der Infrarot- Schnittstelle Richtung Infrarot- Sendestation. Klicke dann auf den OK-Button.
Das Programm schlägt die am häufigsten benötigten Einstellungen vor. Meist wird der RCX mit der Einstellung **Automatic** gefunden.

! Beachte bitte, dass du das **RCX-Command Center** und die Lego-Software nicht gleichzeitig gestartet haben kannst. **!**

Jetzt erscheint die Benutzerschnittstelle, wie unten gezeigt (ohne das Beispiel „1_simple.nqc“).

Das **RCX-Command Center** sieht wie ein Standard- Texteditor (beispielsweise Wordpad) mit dem üblichen Menü aus. Es hat Tasten zum Öffnen Drucken und Bearbeiten von Dateien. Aber es gibt auch einige spezielle Menüs für das Compilieren (= Übersetzen unseres Programmes in eine für den RCX verständliche Computersprache) und das Downloading (= Übertragen unseres Programms in den Speicher des RCX) von Programmen zum Roboter und für das Übertragen von Informationen vom Roboter. Diese benötigen wir erst später. Daher brauchst du sie im Moment nicht zu beachten.
Wir werden ein neues Programm schreiben. Betätige die **NewFile**- Taste, (**neues Programm**) um ein neues, Programmblatt zu erhalten.



2.2. Wir schreiben unser erstes Programm

Schreibe jetzt das folgende Programm. Achte dabei besonders auf die Groß- und Kleinschreibung.

```
task main()
{
  OnFwd(OUT_A);
  OnFwd(OUT_C);
  Wait(400);
  OnRev(OUT_A+OUT_C);
  Wait(400);
  Off(OUT_A+OUT_C);
}
```

Es erscheint dir am Anfang vielleicht ein wenig kompliziert. Schauen wir uns das Programm daher etwas genauer an:
Programme in NQC bestehen aus **Tasks** (engl. = Aufgaben).
Unser Programm hat eine Task, die **`main`** (engl. = Hauptaufgabe) heißt. Jedes Programm muß eine Task `main` haben.
Es ist die Task, die durch den Roboter aufgerufen wird, wenn du die Run- Taste am RCX betätigst.
Du lernst mehr über Tasks in Kapitel 6

Eine Task besteht aus einer Anzahl von Befehlen, auch Anweisungen genannt. Die Befehle einer Task müssen innerhalb der geschweiften Klammern {} stehen. Damit ist klar, daß sie zu dieser Task gehören.
Jede Anweisung endet mit einem Semikolon “ **`;`** “. Auf diese Art ist klar, wo eine Anweisung endet und wo die folgende Anweisung anfängt. Eine Task schaut im allgemeinen wie folgt aus:

```
task main()
{
  Anweisung 1;
  Anweisung 2;
    …
}
```

Unser Programm hat sechs Anweisungen. Wir wollen diese einmal etwas genauer betrachten:

`OnFwd(OUT_A);`
Diese Anweisung erklärt dem Roboter: „schalte Motor A in Richtung vorwärts und schalte ihn ein“. Damit wird der Motor, der mit dem Ausgang A des RCX verbunden ist, angesprochen. Er bewegt sich mit maximaler Geschwindigkeit, es sei denn, dass du zuvor die Geschwindigkeit verändert hast. Wir sehen später, wie du dies tun kannst.

`OnFwd(OUT_C);`
Es ist die gleiche Anweisung, aber jetzt für Motor C. Nach diesen beiden Anweisungen laufen beide Motoren, und der Roboter bewegt sich vorwärts.

`Wait(400);`
Nun ist es Zeit für eine Weile zu warten(wait = engl. warten). Diese Anweisung sagt dem Programm es soll für 4 Sekunden warten.
Der RCX unterbricht den Programmablauf für 4 Sekunden. Die Motoren laufen während dieser Zeit dennoch weiter.
Das Argument, d.h. die Zahl zwischen den Klammern, gibt die Zahl der Zeiteinheiten an. Jede Zeiteinheit beträgt 1/100 einer Sekunde. So kannst du sehr genau steuern, wie lange ein Programm warten soll.
Unser Programm tut also für 4 Sekunden nichts, und der Roboter fährt immer geradeaus weiter.

```
OnRev(OUT_A+OUT_C);
```
Der Roboter hat sich jetzt weit genug bewegt, also erklären wir ihm er soll den Rückwärtsgang einlegen, d.h. rückwärts zu fahren. Beachte bitte, dass wir beide Motoren auf einmal einstellen können, indem wir `OUT_A+OUT_C` als Argument verwenden. Wir hätten die ersten beiden Anweisungen auch auf diese Weise kombinieren können.

```
Wait(400);
```
Wir warten wieder 4 Sekunden.

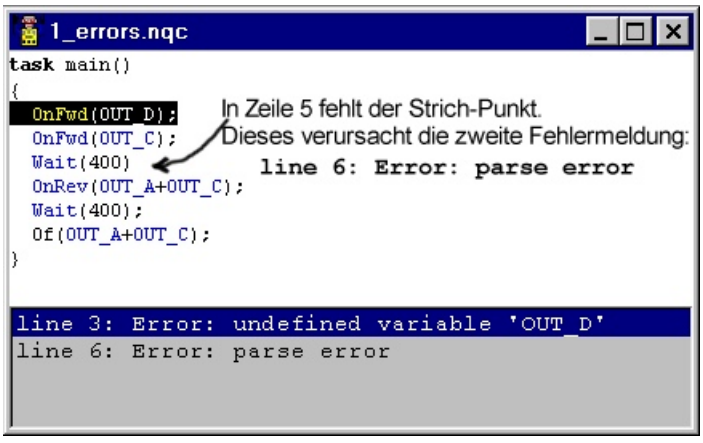
```
Off(OUT_A+OUT_C);
```
Und schließlich schalten wir beide Motoren aus. Das ist das vollständige Programm.
Es schaltet beide Motoren für 4 Sekunden in Richtung vorwärts, dann für 4 Sekunden rückwärts und schaltet sie schließlich aus.
Dir fielen vermutlich die Farben auf, als du das Programm geschrieben hast. Sie erscheinen automatisch, wenn das **RCX Command Center** die Worte erkennt. Alles in blau ist ein Befehl für den Roboter, eine Anweisung für einen Motor oder ein anderer Befehl für den RCX.
Das Wort **`task`** ist in fetter Schrift, weil es ein wichtiges (reserviertes) Wort in NQC ist. Andere wichtige Worte, die in fetter Schrift erscheinen, werden wir später kennen lernen.
Die Farben sind sehr nützlich. Daran kannst du erkennen, dass du keine Fehler beim Schreiben gemacht hast.

2.3. Wir lassen das Programm ablaufen

Sobald du ein Programm geschrieben hast, muß es zunächst compiliert werden (das heißt, es muß in den Code, den der Roboter verstehen und durchführen kann, übersetzt werden). Danach wird es zum Roboter über die Infrarot- Schnittstelle übertragen. (Der Fachmann spricht vom „Downloading“ des Programms). Es gibt eine Taste, die beides automatisch tut (siehe die Abbildung oben). Betätigst du diese Taste, dann wird das Programm nacheinander kompiliert und auf den RCX übertragen. Wenn es Fehler in deinem Programm gibt, werden Sie dir angezeigt (siehe unten.).
Jetzt kannst du dein Programm laufen lassen. Betätige dazu die grüne Run- Taste auf deinem RCX.
Tut der Roboter, was du erwartest? Wenn nicht, sind vermutlich die Leitungen falsch angeschlossen.

2.4. Fehler im Programm ??

Beim Schreiben des Programms ist es wahrscheinlich, dass du einige Fehler machst. Der Compiler bemerkt diese Fehler und teilt sie dir am unteren Rand des Fensters mit, wie in der folgenden Abbildung:



Er wählt automatisch den ersten Fehler aus (hier der falsche Name des Motors).
Erklärungen zu den Fehlermeldungen findest du im Anhang.
Wenn es mehrere Fehler gibt, kannst du die Fehlermeldungen anklicken, um zu ihnen zu gelangen.
Beachte bitte, dass häufig Fehler am Anfang des Programms andere Fehler an anderen Plätzen verursachen. Behebe also zunächst die ersten Fehler und compiliere dann das Programm erneut.
Beachte auch, dass dir die Farb- Kodierung viel hilft, um Fehler zu erkennen. Beispielsweise in der letzten Zeile schrieben wir “**`Of`**” anstatt “**`Off`**”. Weil dieses ein unbekannter Befehl ist, wird er nicht blau gefärbt.

! Die häufigsten Fehler sind falsch gesetzte Klammern und vergessene Strickpunkte am Ende der Befehlszeile.
Wenn beispielsweise die Meldung **`line 6 : Error: parse error`** erscheint, so schau doch zuerst in die Zeile zuvor, ob dort nicht der Strich-Punkt fehlt! !

Es gibt auch Fehler, die nicht vom Compiler gefunden werden. Wenn wir OUT_B geschrieben hätten, wäre der Fehler unbemerkt geblieben, weil dieser Motor existiert, obwohl wir ihn nicht im Roboter benutzen.
Wenn dein Roboter sich also anders verhält, als du erwartet hast, so ist ein Fehler in deinem Programm die wahrscheinlichste Ursache.

2.5. Ändern der Geschwindigkeit

Wie du vielleicht bemerkt hast, bewegte sich dein Roboter ziemlich schnell. Im RCX ist zunächst die maximale Geschwindigkeit voreingestellt. Um die Geschwindigkeit zu ändern, mußt du den Befehl `SetPower( )` verwenden.
Die Geschwindigkeit ist eine Zahl zwischen 0 und 7. 7 ist die schnellste, 0 die langsamste, wobei sich der Roboter langsam weiterbewegt.
Die Geschwindigkeit hängt dabei wesentlich vom Antrieb deines Roboters ab. Ist dieser schwergängig, dann erreichst du beispielsweise mit 5 eine niedrigere Geschwindigkeit, als mit einem leichtgängigen Antrieb bei 3.

Ist hier eine neue Version unseres Programms, mit dem sich der Roboter langsamer bewegt:

```
task main()
{
  SetPower(OUT_A+OUT_C,2);
  OnFwd(OUT_A+OUT_C);
  Wait(400);
  OnRev(OUT_A+OUT_C);
  Wait(400);
  Off(OUT_A+OUT_C);
}
```

2.6. Zusammenfassung

In diesem Kapitel hast du dein erstes Programm in NQC mit dem **RCX-Command Center** geschrieben.
Du hast gelernt, wie man ein Programm compiliert (= in die für den RCX verständliche Sprache übersetzt). Außerdem hast du gelernt, wie du dein Programm zum Roboter downloaden (=übertragen) kannst, und wie du dann deinen Roboter startest.
Aber mit dem **RCX-Command Center** kannst du noch viel mehr tun. Um das herauszufinden, lies bitte die Unterlagen RcxCC.doc und NQC_Guide.doc, die beim **RCX-Command Center** dabei sind. (leider noch nicht in deutsch)

Dieser Leitfaden beschäftigt hauptsächlich mit der Sprache NQC. Eigenschaften des RCX und **RCX-Command Center** werden nur beschrieben, wenn du sie wirklich benötigst.
Du hast auch einige wichtige Merkmale der Sprache NQC kennengelernt. Als erstes hast du erfahren, daß jedes Programm eine **`task`** `main()` hat, welche immer beim Starten des Programms als erstes aufgerufen wird.
Außerdem hast du die vier wichtigsten Bewegungsbefehle: `OnFwd()`, `OnRev()`, `SetPower()` und `Off()` kennengelernt, sowie die Anweisung `Wait()`.

3. Ein interessanteres Programm

Unser erstes Programm war nichts Besonderes. Wir wollen es daher etwas interessanter gestalten.
Wir tun dies in den folgenden Schritten und lernen dabei einige Besonderheiten der Sprache NQC kennen.

3.1. Der Roboter dreht sich

Du kannst deinen Roboter dazu veranlassen sich zu drehen, indem du einen der beiden Motoren stoppst oder seine Drehrichtung umkehrst. Hier ist ein Beispiel. Tippe es ein, downloade es zu deinem Roboter und laß‘ es laufen.
Er sollte etwas nach vorne fahren und dann eine 90-Grad- Drehung nach rechts durchführen.

```
task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(200);
  OnRev(OUT_C);
  Wait(160);
  Off(OUT_A+OUT_C);
}
```

Wahrscheinlich mußt du die Zahl 160 im zweiten Befehl `Wait()` etwas verändern, um eine exakte 90-Grad-Drehung zu erreichen. Dieses hängt vom Antrieb deines Roboters, dem Ladezustand der Batterien oder Akkus und von der Art der Oberfläche ab, auf welcher der Roboter läuft. Anstatt, dieses im Programm zu ändern, ist es einfacher, einen Namen für diese Zahl zu verwenden.
In NQC kannst du konstante Werte (= immer gleichbleibende Werte) definieren (= festlegen), wie im folgenden Programm gezeigt.

```
#define MOVE_TIME   200
#define TURN_TIME   160

task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(MOVE_TIME);
  OnRev(OUT_C);
  Wait(TURN_TIME);
  Off(OUT_A+OUT_C);
}
```

Die ersten beiden Zeilen definieren zwei Konstanten. Diese können vom RCX während des gesamten Programmablaufs verwendet werden.
Die Verwendung von Konstanten ist aus zwei Gründen zu empfehlen:

- Erstens wird das Programm übersichtlicher und einfacher lesbar, da du deinen Konstanten logische Namen geben kannst und da **RCX-Command Center** den Konstanten eine eigene Farbe gibt.
  In unserem Beispiel heißen die Namen `MOVE_TIME` = Fahrzeit und `TURN_TIME` = Drehzeit.
- Zweitens ist die Verwendung von Konstanten besonders bei größeren Programmen sinnvoll, da du diese Werte bereits am Beginn deines Programmes festlegen kannst, wo du sie viel leichter findest, als irgendwo zwischen den vielen anderen Anweisungen.

Wie du in Kapitel 6 sehen wirst, kannst du außer Konstanten auch andere Dinge definieren.

3.2. Befehle wiederholen

Wir wollen jetzt ein Programm schreiben, das den Roboter in einem Quadrat fahren läßt.
Fahren in einem Quadrat bedeutet: der Roboter fährt ein Stück gerade, macht eine 90º-Drehung, fährt wieder ein Stück gerade, macht wieder eine 90º-Drehung , usw. Wir könnten die in unserem vorherigen Programm verwendeten Befehlszeilen viermal wiederholen, aber dieses kann man mit der **`repeat`**- Anweisung viel einfacher gestalten.
(repeat = engl. für wiederholen)

```
#define MOVE_TIME   200
#define TURN_TIME   160

task main()
{
  repeat(4)
  {
    OnFwd(OUT_A+OUT_C);
    Wait(MOVE_TIME);
    OnRev(OUT_C);
    Wait(TURN_TIME);
  }
  Off(OUT_A+OUT_C);
}
```

Falls dein Roboter kein schönes Quadrat gefahren ist, verändere doch die Werte für `MOVE_TIME` und `TURN_TIME` in den ersten beiden Zeilen.

Die Zahl hinter der **`repeat`**- Anweisung in Klammern ( ), zeigt an, wie oft etwas wiederholt werden muß.
Die Anweisungen, die wiederholt werden müssen, werden zwischen Klammern { } gesetzt, genau so wie die Anweisungen in einer **`task`**.
Beachte bitte, dass im oben gezeigten Programm auch die Anweisungen einrücken. Dieses ist nicht notwendig, aber es macht das Programm lesbarer. Als abschließendes Beispiel lassen wir den Roboter 10mal im Quadrat fahren.

Hier ist das Programm:

```
#define MOVE_TIME   200
#define TURN_TIME   160

task main()
{
  repeat(10)
  {
    repeat(4)
    {
      OnFwd(OUT_A+OUT_C);
      Wait(MOVE_TIME);
      OnRev(OUT_C);
      Wait(TURN_TIME);
    }
  }
  Off(OUT_A+OUT_C);
}
```

Es gibt hier eine **repeat** - Anweisung innerhalb der anderen. Man nennt dies eine " verschachtelte " **repeat**- Anweisung. du kannst **repeat** - Anweisungen verschachteln so viele du willst. Beachte bitte die Klammern {} und Einrückungen im Programm. Die **task** `main ()` beginnt mit der ersten Klammer "{" und endet mit der letzten "}".
Die erste **repeat** - Anweisung beginnt mit der zweiten Klammer "{" und endet mit der vorletzten "}". Die zweite, verschachtelte, **repeat** - Anweisung beginnt mit der dritten Klammer "{" und endet mit der drittletzten "}".
Du siehst, Klammern { } werden immer paarweise verwendet und die Befehle dazwischen rücken wir etwas ein.

3.3. Kommentare hinzu fügen

Kommentare =Bemerkungen, Anmerkungen, Hinweise.
Um dein Programm lesbarer zu gestalten, ist es sinnvoll Kommentare hinzuzufügen. Wann immer du die beiden Schrägstriche // verwendest, wird der Rest dieser Zeile ignoriert, also nicht als Befehl für den RCX verstanden, und kann für Kommentare verwendet werden. Lange Kommentare werden zwischen /* und */ gesetzt und können sich über mehrere Zeilen erstrecken. Kommentar wird im **RCX-Command Center** grün gefärbt.
Dein Programm könnte wie folgt ausschauen:

```
/*  10 Quadrate

    by Mark Overmars

Dieses Programm veranlaßt den Roboter 10 Quadrate zu fahren
*/

#define MOVE_TIME   200     // Zeit für die Geradeausfahrt
#define TURN_TIME   160     // Zeit für die 90°-Kurve

task main()
{
  repeat(10)                // Fahre 10 Quadrate
  {
    repeat(4)               // Fahre die vier Ecken und Seiten
    {
      OnFwd(OUT_A+OUT_C);
      Wait(MOVE_TIME);
      OnRev(OUT_C);
      Wait(TURN_TIME);
    }
  }
  Off(OUT_A+OUT_C);         // Schalte nun die Motoren ab
}
```

3.4. Zusammenfassung

In diesem Kapitel haben wir die **repeat** - Anweisung und die Verwendung von Kommentaren gelernt.
Außerdem lernten wir die Verwendung verschachtelter Anweisungen kennen, und wie man zusammengehörende Anweisungen durch Klammern { } und Einrücken als zusammengehörend kenntlich macht.
Damit kannst du deinen Roboter schon dazu bringen die verschiedensten Wege zu fahren.

Übe das doch ein wenig und verändere dein Programm ein paar mal, bevor du im nächsten Kapitel weitermachst.

4. Wir verwenden Variablen

Variablen sind ein sehr wichtiger Teil jeder Programmiersprache. Sie sind Speicherstellen im Programm, die während des Programmablaufs verschiedene Werte annehmen können. Wir können diese Werte an den unterschiedlichsten Stellen im Programm verwenden, und wir können sie ändern. Wir wollen die Verwendung von Variablen an einem Beispiel lernen.

4.1. Unser Roboter fährt eine Spirale

Nehmen wir an, wir möchten das oben gezeigte Programm anpassen, so dass der Roboter eine spiralförmige Bahn fährt. Dieses kann erzielt werden, indem man die Zeit, die das Programm bis zur Ausführung des nächsten Programmschrittes wartet, jedesmal etwas vergrößert. Das heißt, wir werden den Wert von MOVE_TIME jedes mal etwas erhöhen. Aber wie können wir dies tun? MOVE_TIME ist eine Konstante und Konstanten können nicht geändert werden. Wir benötigen statt dessen eine Variable. Variablen können in NQC leicht definiert werden. Du kannst bis zu 32 Variablen in deinem Programm verwenden, und du kannst (oder besser gesagt du mußt) jeder von ihnen einen unterschiedlichen Namen geben.
Ist hier das Spiralen- Programm.

```
#define TURN_TIME   160

int move_time;               // definiert die Variable
task main()
{
  move_time = 100;           // weist der Variablen einen Startwert
zu
  repeat(50)
  {
    OnFwd(OUT_A+OUT_C);
    Wait(move_time);        // verwendet die Variable für die
Pausenlänge
    OnRev(OUT_C);
    Wait(TURN_TIME);
    move_time += 20;         // erhöht den Wert der Pause um 10
  }
  Off(OUT_A+OUT_C);
}
```

Die interessanten Zeilen werden mit Kommentaren erklärt. Zuerst definieren wir eine Variable, indem wir das Schlüsselwort **`int`** gefolgt von einem Namen schreiben. Normalerweise benutzen wir Kleinschreibung für Namen von Variablen und Großschreibung für Konstanten. Das ist nicht unbedingt notwendig, hilft uns aber im Programm diese beiden Werte zu unterscheiden. Der Name muß mit einem Buchstaben beginnen, aber er kann auch Ziffern und das Unterstrich- Zeichen (erreichst du durch Umschalt + Minus) enthalten. Andere Symbole, auch die deutschen Umlaute, wie ä, ö und ü, sind nicht erlaubt. Das gilt auch für die Bezeichnungen von Konstanten, tasks usw. Das fett geschriebene **`int`** steht für Integer und bedeutet ganzzahlig. Nur ganze Zahlen, also ohne Nachkommastellen, können in ihr gespeichert werden.
In der zweiten interessanten Zeile weisen wir der Variable den Wert 20 zu. Von diesem Moment an, wann immer die Variable verwendet wird, steht sie für 20. Jetzt folgt die **`repeat`**- Schleife (du weißt noch? repeat = wiederholen), in der wir die Variable verwenden. Mit **`Wait`**`(move_time)` sagen wir dem Programm, wie lange es bis zur Ausführung des nächsten Programmschrittes warten soll. Dieser Wert wird am Ende der Schleife mit dem Befehl `movetime += 20` um 20 erhöht, so dass der Programmablauf beim ersten mal 100 ticks (100 / 100 Sekunden = 1 Sekunde) unterbrochen wird, beim zweiten mal 120, beim dritten mal 140, usw.

Neben addieren zu einer Variable können wir eine Variable auch mit einer Zahl multiplizieren mit * =, subtrahieren mit - = und durch das Verwenden von / = teilen. Beim Teilen wird das Ergebnis zur nächsten ganzen Zahl gerundet.(Du weißt noch? Mit **`int`** können nur ganze Zahlen verwendet werden.) Du kannst verschiedene Variablen miteinander addieren, oder andere mathematische Operationen ausführen. Hier sind einige Beispiele:

```
int aa;               // Definition der Variablen aa
int bb, cc;           // gleichzeitige Definition der Variablen bb und cc
task main()
{
  aa = 10;            // weise der Variablen aa den Wert 10 zu
  bb = 20 * 5;        // weise bb den Wert 20 mal 5 = 100 zu
  cc = bb;            // sezte cc gleich bb, cc wird also ebenfalls 100
  cc /= aa;           // berechne cc / aa und weise das Ergenis cc zu, cc = 100/10 = 10
  cc -= 1;            // ziehe von cc 1 ab und weise das Ergebnis cc zu, cc wird also 9
  aa = 10 * (cc + 3); // berechne 10 * (cc + 3), aa wird also 10 * (9 + 3) = 120
}
```

Beachte, dass wir in einer Zeile mehrere Variablen definieren können, wie in Zeile zwei gezeigt. Wir könnten auch alle drei Variablen in einer Zeile definieren.

4.2. Zufallszahlen

In allen oben genannten Programmen haben wir genau bestimmt, was der Roboter tun sollte. Aber die Sache ist viel interessanter, wenn der Roboter Dinge tut, die wir nicht vorhersagen können. Wir möchten, dass der Roboter zufällige Bewegungen ausführt. In NQC kannst du Zufallszahlen bilden, so wie bei einem Würfelspiel.
Vielleicht kannst du deinen Roboter ja so programmieren, dass er dir beim nächsten „Mensch-ärgere-dich-nicht“-Spiel die Würfel ersetzt (das Programm ist aber sicher ziemlich schwer).Wie du die Augenzahl anzeigen lassen kannst, kannst du in Kapitel 12.2 nachlesen.
Das folgende Programm verwendet Zufallszahlen und läßt den Roboter zufällige Bewegungen ausführen. Er fährt dann eine unbestimmte Zeit vorwärts und macht dann eine zufällige Wendung.

```
int move_time, turn_time;

task main()
{
  while(true)
  {
    move_time = Random(60);
    turn_time = Random(40);
    OnFwd(OUT_A+OUT_C);
    Wait(move_time);
    OnRev(OUT_A);
    Wait(turn_time);
  }
}
```

Das Programm definiert zwei Variablen und weist ihnen dann die Zufallszahlen zu.
`Random(60)` bedeutet eine Zufallszahl zwischen 0 und 60. Diese Zahlen verändern sich ständig. Mit der Verwendung der Variablen `move_time` und `Wait (move_time)` konnten wir die ungünstige Schreibweise `Wait(Random(60))` vermeiden.
Du lernst hier auch eine neue Möglichkeit kennen, wie du Schleifen programmieren kannst.
Anstatt der **`repeat`**- Anweisung haben wir hier **`while`** (**`true`**) verwendet. **`while`** ist englisch und bedeutet “während”, “so lange wie”.
Die **`while`**- Anweisung wiederholt also die Anweisungen unterhalb so lange, wie die Bedingung zwischen den Klammern ( ) zutreffend ist. Das spezielle Wort **`true`** bedeutet “wahr” bzw. “immer zutreffend”, also werden die Anweisungen zwischen den Klammern { } für immer wiederholt, wir haben eine so genannte Endlos- Schleife.
In Kapitel 4 lernst du mehr über die **`while`**- Anweisung.

4.3. Zusammenfassung

In diesem Kapitel hast du etwas über die Verwendung von Variablen gelernt. Variablen sind sehr nützlich, aber wegen des RCX gibt es kleine Einschränkungen. Es können höchstens 32 von ihnen definiert werden, und sie können nur ganze Zahlen speichern. Aber für viele Roboter- Aufgaben ist dieses gut genug. Du hast auch gelernt, wie du deinem Roboter mit Hilfe der Zufallszahlen ein unvorhersehbares Verhalten geben kannst. Schließlich sahen wir, wie man mit der **`while`** - Anweisung eine Endlos- Schleife bilden kann, die sich für immer wiederholt.

5. Steuerbefehle

In den vorhergehenden Kapiteln lernten wir die **`repeat`** - und **`while`** – Befehle kennen. Diese Anweisungen steuern die Art und Weise, wie die anderen Anweisungen im Programm durchgeführt werden. Sie werden " Steuerbefehle" genannt.
In diesem Kapitel lernen wir einige andere Steuerbefehle.

5.1. Die `if` - Anweisung

Manchmal möchte man, dass ein bestimmter Teil eines Programms nur in bestimmten Situationen ausgeführt wird.
In diesem Fall findet die **`if`**- Anweisung Anwendung. Betrachten wir das an einem Beispiel:
Wir ändern wieder das Programm, das wir bis jetzt bearbeitet haben, aber mit einer Neuerung. Wir wollen, dass der Roboter geradeaus fährt und dann plötzlich eine Links- oder Rechtsdrehung durchführt. Dazu benötigen wir wieder unsere Zufallszahlen. Wir wählen eine Zufallszahl zwischen 0 und 1 aus, d.h. sie ist entweder 0 oder 1.
Wenn die Zahl 0 ist, soll der Roboter rechts herum fahren, andernfalls links herum.
Ist hier das Programm:

```
#define MOVE_TIME   200
#define TURN_TIME   160

task main()
{
  while(true)
  {
    OnFwd(OUT_A+OUT_C);
    Wait(MOVE_TIME);
    if (Random(1) == 0)
    {
      OnRev(OUT_C);
    }
    else
    {
      OnRev(OUT_A);
    }
    Wait(TURN_TIME);
  }
}
```

Die **`if`**- Anweisung ähnelt der **`while`** – Anweisung. Wenn die Bedingung zwischen den Klammern ( ) zutrifft, wird der Teil zwischen den Klammern { } durchgeführt. Andernfalls wird der Teil zwischen den Klammern { } nach dem Wort **`else`** durchgeführt.
Betrachten wir die Bedingung innerhalb der Klammern ( ) nach der **`if`**- Anweisung etwas genauer:
Sie heißt `Random(1) == 0`. Dies heißt, dass `Random(1)` gleich 0 sein muß, damit die Bedingung zutrifft. Vielleicht wunderst du dich , warum wir `==` benutzen, anstatt `=`. Damit soll die Bedingung von einer Anweisung unterschieden werden, bei der einer Variablen ein Wert zugewiesen wird.
Werte können auf unterschiedliche Weise unterschieden werden:

`==`	ist genau gleich
`<`	kleiner als
`<=`	kleiner oder gleich
`>`	größer als
`>=`	größer oder gleich
`!=`	ungleich

Du kannst die Bedingungen mit `&&` untereinander kombinieren, was “und” bedeutet, oder mit `||`, was “oder” bedeutet. (Den |-Strich errreichst du mit der Tastenkomination Alt Gr und ><)
Hier sind einige Beispiele für Bedingungen:

`true`	immer wahr
`false`	nie wahr
`ttt != 3`	wahr, wenn ttt ungleich 3 ist
`(ttt >= 5) && (ttt <= 10)`	wahr, wenn ttt zwischen 5 und 10 liegt
`(aaa == 10) || (bbb == 10)`	wahr, wenn aaa oder bbb (oder beide) genau 10 sind

Beachte bitte, dass, die **`if`**- Anweisung aus zwei Teilen besteht. Die eine Hälfte sofort nach der **`if`**- Bedingung, die ausgeführt wird, wenn die Bedingung zutreffend ist, und die andere Hälfte nach dem **`else`**, welche durchgeführt wird, wenn die Bedingung falsch ist. Das Schlüsselwort **`else`** und die Anweisungen nach ihm müssen aber nicht unbedingt sein. Sie können weggelassen werden, wenn es nichts gibt zu tun, falls die Bedingung falsch ist.

5.2. Die `do`- Anweisung

Ein weiterer Steuerbefehl ist die **`do`**- Anweisung. Sie hat das folgenden Aufbau:

```
do
{
  Befehle;
}
while (Bedingung);
```

Die Anweisungen zwischen den Klammern{ } nach **`do`** werden so lange durchgeführt, wie die Bedingung in den Klammern `()` nach dem Wort **`while`** zutreffend ist. Die Bedingung hat den gleichen Aufbau wie bei der **`if`** - Anweisung, die oben beschrieben wird. Ist hier ein Beispiel eines Programms.
Der Roboter fährt höchstens 20 Sekunden lang (**`while`** `(total_time < 2000)`) zufällig herum.
2000 bedeutet 2000 / 100 = 20 Sekunden.

```
int move_time, turn_time, total_time;

task main()
{
  total_time = 0;            // Setzt Startwert = 0
  do
  {
    move_time = Random(100); //fährt höchstens eine Sekunde lang gerade
    turn_time = Random(100); //macht immer nur kleine Kurven
    OnFwd(OUT_A+OUT_C);
    Wait(move_time);
    OnRev(OUT_C);
    Wait(turn_time);
    total_time += move_time; total_time += turn_time; //berechnet neue
Zeit
  }
  while (total_time < 2000);  //überprüft, ob Gesamtzeit erreicht ist
  Off(OUT_A+OUT_C);
}
```

Beachte bitte, dass in diesem Beispiel zwei Befehle in eine Zeile geschrieben wurden, nämlich:

total_time += move_time; total_time += turn_time;

Dieses ist erlaubt. du kannst beliebig viel Anweisungen in eine Zeile schreiben, nur mußt du sie immer durch einen Strich-Punkt ";" voneinander trennen. Aber für die Lesbarkeit des Programms ist es nicht sehr sinnvoll.
Beachte bitte auch, dass die **`do`**- Anweisung sich fast wie die **`while`** –Anweisung verhält, aber mit einem Unterschied:
Die **`while`**- Anweisung prüft die Bedingung innerhalb der Klammern ( ) vor dem Abarbeiten der Befehle innerhalb der Klammern { }, während die **`do`**- Anweisung dies erst danach tut. Das heißt, dass die Befehle { } nach der **`while`**- Anweisung unter Umständen nie abgearbeitet werden, nämlich dann, wenn die Bedingung ( ) nie zutrifft.
Bei der **`do`**- Anweisung werden die Befehle { } zumindest einmal abgearbeitet, da die Bedingung ( ) ja erst danach überprüft wird.

5.3. Zusammenfassung

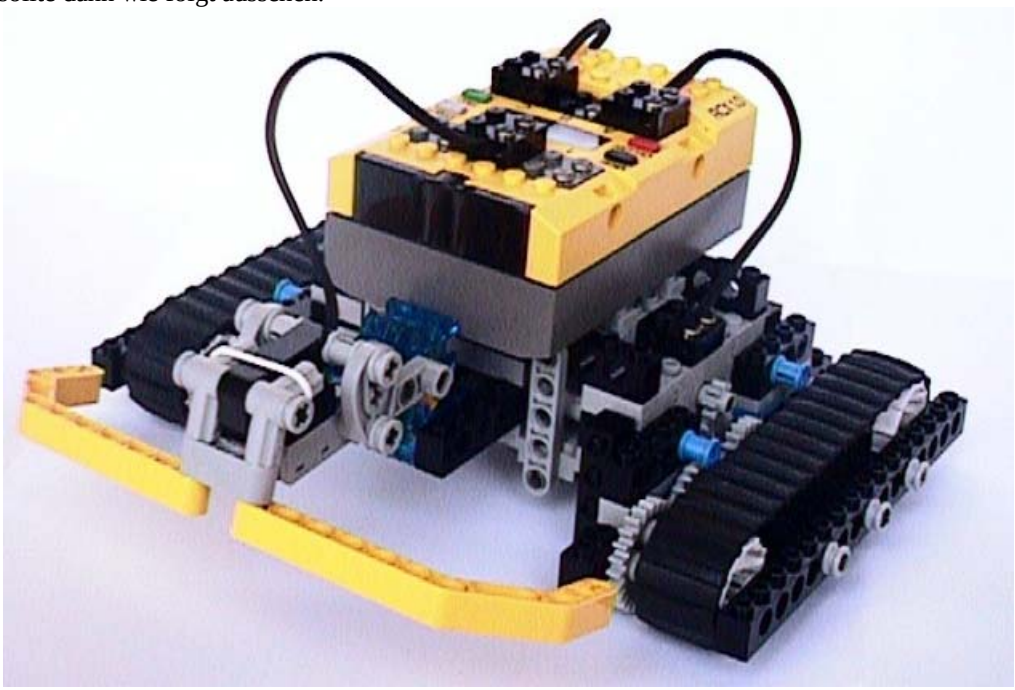
In diesem Kapitel haben wir zwei neue Steuerbefehle kennen gelernt: die **`if`** - Anweisung und die **`do`** - Anweisung. Zusammen mit der **`repeat`** - Anweisung und der **`while`** - Anweisung sind sie diejenigen Befehle, welche die Art und Weise steuern, in der das Programm durchgeführt wird.
Es ist sehr wichtig, dass du sie verstanden hast. Probiere diese Befehle an ein paar eigenen Programmen aus, bevor du dich mit den weiteren Befehlen von NQC beschäftigst.

Verändere das oben gezeigte Programm doch einmal so, dass dein Roboter nicht nur Rechtskurven fährt, sondern zufällig zwischen rechts und links wechselt. Schaue dir dazu das Programm eine Seite vorher im Kapitel „4.1 Die `if`-Anweisung" nochmals an.

In diesem Kapitel lernten wir auch, dass wir mehrere Anweisungen in eine Zeile schreiben können. Allerdings geht dies auf Kosten der Lesbarkeit des Programmes.

6. Sensoren

Sensoren sind Fühler, wie für uns Menschen die Augen, die Nase oder die Hände, mit denen wir Dinge ertasten können. Das tolle an unserem Lego-Roboter ist, dass wir Sensoren an ihn anschließen können, und dass wir den Roboter auf die Sensoren reagieren lassen können.
Bevor wir mit dem Einbinden von Sensoren in ein NQC - Programm beginnen können, müssen wir unseren Roboter etwas umbauen, indem wir einen Tast - Sensor hinzufügen. Baue zu diesem Zweck den Stoßfänger mit Einzelsensor an deinen Roboter an, so wie es auf den Seiten 26 bis 29 deiner **CONSTRUCTOPEDIA** beschrieben ist.
Dein Roboter sollte dann wie folgt aussehen.



Schließe den Taster an Eingang 1 des RCX an.

6.1. Abfragen eines Sensors

Wir wollen mit einem einfachen Programm beginnen, bei dem der Roboter geradeaus fährt, bis er gegen ein Hindernis stößt:

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH); //definiert Sensor an Eingang 1
  OnFwd(OUT_A+OUT_C);               //der Roboter fährt geradeaus
  until (SENSOR_1 == 1);            //bis er gegen etwas stößt,
  Off(OUT_A+OUT_C);                 //dann wird er abgeschaltet
}
```

In diesem Programm gibt es zwei Zeilen mit neuen Befehlen, die wir uns einmal genauer ansehen wollen:
Die erste Zeile des Programms sagt dem RCX welche Art von Sensor wir verwenden. `SENSOR_1` ist die Nummer des Eingangs, mit dem wird den Sensor verbunden haben. Die beiden anderen Eingänge heißen `SENSOR_2` und `SENSOR_3`. `SENSOR_TOUCH` sagt dem RCX, dass wir einen Taster verwenden (touch = engl. berühren). Für den Lichtsensor würden wir `SENSOR_LIGHT` verwenden.
Nachdem wir den Sensor- Typ festgelegt haben, schaltet das Programm beide Motoren ein und der Roboter fährt vorwärts. Der Befehl **`until`** (engl. = so lange bis) ist sehr leistungsfähig. Hier unterbricht der Programmablauf, bis die Bedingung innerhalb der Klammern ( ) zutrifft.

Die hier gezeigte Bedingung besagt, dass der Wert von `SENSOR_1` = 1 sein muß. Das bedeutet, dass der Sensor gedrückt wurde. Solange der Sensor nicht betätigt wurde, ist der Wert = 0. Daher wartet das Programm hier, bis der Sensor gedrückt wurde, und weil die beiden Motoren zuvor eingeschaltet wurden, läuft der Roboter geradeaus weiter. Sobald der Taster betätigt wird, wird der Wert von `SENSOR_1` = 1 und damit die Bedingung innerhalb der Klammern ( )wahr, der Programmablauf wird fortgesetzt und kommt zum Befehl `Off(OUT_A+OUT_C);` der beide Motoren abschaltet. Damit endet auch das Programm. Du erkennst dies daran, dass das Männchen in der Anzeige des RCX stehen bleibt.

6.2. Reagieren auf einen Berührungssensor

Wir wollen nun, dass unser Roboter Hindernissen ausweicht. Wann immer der Roboter an ein Hindernis stößt, soll er etwas zurückfahren, dann eine Wendung machen und danach geradeaus weiterfahren. Ist hier das Programm:

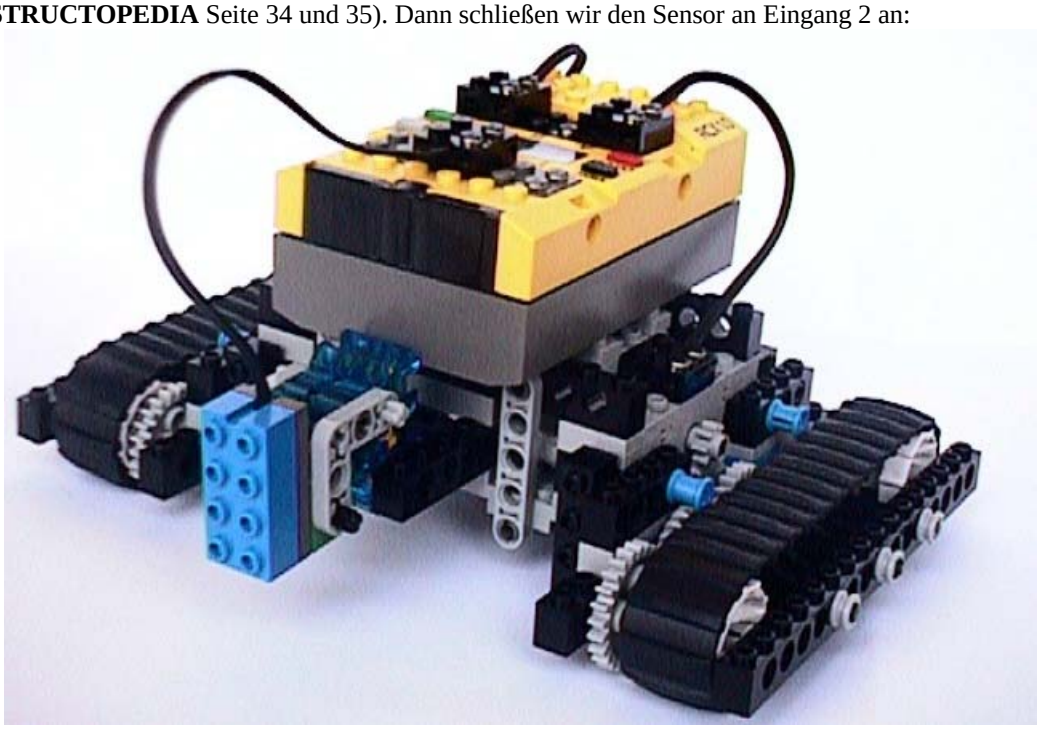
```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH);  //definiert den Eingang
  OnFwd(OUT_A+OUT_C);                //schaltet die Motoren ein
  while (true)                       //bildet eine Endlosschleife
  {
    if (SENSOR_1 == 1)               //fragt ab, ob Taster gedrückt wurde
    {                                //wenn ja, kommen hier {  } die
      OnRev(OUT_A+OUT_C); Wait(60);  //Befehle für das Ausweichmanöver.
      OnFwd(OUT_A); Wait(60);        //Danach kommt der nächste Durch-
      OnFwd(OUT_A+OUT_C);            //lauf der Schleife.
    }
  }
}
```

Wie im vorhergehenden Beispiel, sagen wir dem RCX als erstes, welcher Sensor angeschlossen wurde. Dann werden beide Motoren in Richtung vorwärts gestartet (`OnFwd`) und der Roboter fährt geradeaus los. Dann kommt eine Endlos- Schleife (**`while`** (**`true`**)). Dadurch wird erreicht, dass der Sensor ständig abgefragt wird.
Wird die **`if`** – Bedingung `SENSOR_1 == 1` erfüllt, das heißt der Taster wurde gedrückt, weil der Roboter gegen ein Hindernis gefahren ist, so fährt er 0,6 Sekunde rückwärts, dann macht er für 0,6 Sekunde eine Rechtsdrehung und fährt anschließend wieder geradeaus weiter.
Beachte bitte auch wie hier die Rechtsdrehung programmiert wurde. Während sich der rechte Motor noch vom Zurückfahren rückwärts dreht, wird der linke Motor vorwärts geschaltet. Dadurch dreht sich der Roboter auf der Stelle.

6.3. Lichtsensoren

Mit dem Lego MindStorms- System hast du neben zwei Tastern auch einen Lichtsensor erhalten.
Der Lichtsensor erfaßt die Lichtmenge, die an seiner Vorderseite neben der roten Leuchtdiode (LED) einfällt. Daneben strahlt er über die LED selbst Licht ab. Damit kann der Lichtsensor Fremdlicht, also von der Zimmerbeleuchtung oder das von einem nahen Gegenstand zurückgeworfene (reflektierte) Licht der LED erfassen. Du kannst damit deinen Roboter einer dunklen Markierung auf dem Fußboden, etwa der Bahn auf dem mitgelieferten Poster, folgen lassen.

Das wollen wir nun auch in unserem Beispiel tun. Dazu müssen wir zunächst den Lichtsensor an unseren Roboter anbringen (siehe **CONSTRUCTOPEDIA** Seite 34 und 35). Dann schließen wir den Sensor an Eingang 2 an:



Nun benötigen wir nur noch unsere Bahn aus dem Lego-MindStorms- Kasten und natürlich das Programm.
Das basiert auf folgenden Gedanken: sobald der Roboter die Bahn verläßt, wird vom nun hellen Untergrund mehr Licht zurückgeworfen und der Roboter muß seine Bahn korrigieren. Es funktioniert aber nur, wenn der Roboter im Uhrzeigersinn, also rechts herum fährt.

```
#define THRESHOLD 40                  //definiert Grenzwert für Helligkeit

task main()
{
  SetSensor(SENSOR_2,SENSOR_LIGHT); //definiert Eingang 2 als Lichtsensor
  OnFwd(OUT_A+OUT_C);               //schaltet beide Motoren auf vorwärts
  while (true)                      //unsere bekannte Endlosschleife
  {
    if (SENSOR_2 > THRESHOLD)       //vergleicht Sensorwert mit Grenzwert
    {
      OnRev(OUT_C);                 //schaltet rechten Motor auf rückwärts
      until (SENSOR_2 <= THRESHOLD);//bis dunkle Bahn wieder erreicht ist
      OnFwd(OUT_A+OUT_C);           //dann geht's wieder geradeaus
    }
  }
}
```

In der allerersten Zeile definieren wir mit #define THRESHOLD 40 einen Vergleichswert für die Helligkeit.
Dieser ist hier 40. Es kann sein, dass du ihn, je nach Lichtverhältnissen in deinem Zimmer, ändern mußt.

Dann teilen wir dem RCX mit, dass wir an Eingang 2 einen Helligkeits- Sensor angeschlossen haben, und beide Motoren werden in Vorwärtsrichtung gestartet.
Danach kommt die uns schon bekannte Endlosschleife (**while** (**true**)).
Nun wird der Sensor abgefragt. Wann immer der Wert größer als 40 ist (= heller Untergrund), schalten wir den Motor C in den Rückwärtsgang, bis der Helligkeitswert wieder unter 40 gesunken ist (=dunkler Untergrund).

Wenn du das Programm ausprobierst, wirst du erkennen, dass die Bewegung recht ruckelig ist. Setze einen **`Wait`**`(10)` – Befehl vor den **`until`**- Befehl, dann wirst du sehen, dass die Bewegungen etwas ruhiger werden.
Beachte bitte auch, dass das Programm nur für die Fahrt im Uhrzeigersinn geeignet ist. Du kannst es ja einmal für die Fahrt entgegen dem Uhrzeigersinn programmieren.
Für beliebige Strecken ist ein viel komplizierteres Programm nötig. Wenn du mit deinem Lego-Roboter und NQC vertraut bist, kannst du es ja einmal versuchen!

6.4. Zusammenfassung

In diesem Kapitel hast du gelernt, wie du Berührungssensoren und Lichtsensoren an deinem Roboter einsetzen kannst. Wir lernten auch den **`until`**- Befehl kennen, der gerade bei der Verwendung von Sensoren sehr nützlich ist.

Bevor du in dieser Anleitung weiter arbeitest, solltest du das bisher gelernte etwas üben.
Baue deinen Roboter doch etwas um, und probiere die verschieden Möglichkeiten, die du nun mit den Sensoren hast.
Du hast nun so viel gelernt, dass du schon ganz schön schwierige Aufgaben lösen kannst.
Ich habe hier ein paar Ideen:

- Baue eine Roboter mit zwei Berührungssensoren, die ihn nach links oder rechts ausweichen lassen, je nachdem welcher Sensor berührt wurde.
- Schreibe ein Programm, welches verhindert dass dein Roboter über die Tischkante hinausfährt und damit vom Tisch fällt.
- Baue einen Roboter, der sich nur innerhalb des mit dem dunklen Rand begrenzen Feldes bewegt.
- Baue eine Roboter ...

7. Tasks (Aufgaben) und Unterprogramme

Bis jetzt bestanden alle unsere Programme aus genau einer **`Task`**. Aber NQC- Programme können aus mehreren Tasks bestehen. Es ist auch möglich, Programmabschnitte in sogenannte Unterprogramme einzusetzen, die du von den unterschiedlichsten Stellen deines Programmes aus aufrufen kannst. Das Verwenden von Tasks und von Unterprogrammen macht dein Programm übersichtlicher und vom Umfang her kleiner. In diesem Kapitel betrachten wir die verschiedenen Möglichkeiten.

7.1. Tasks

Ein NQC- Programm kann aus bis zu 10 Tasks bestehen. Jede Task hat einen Namen. Eine Task muß **`main`** ( ) heißen. Diese wird ausgeführt, wenn die grüne **Run- Taste** auf dem RCX betätigt wird. Die anderen Tasks werden nur durchgeführt, wenn eine laufende Task sie aufruft. Von diesem Moment an laufen beide Tasks gleichzeitig (die aufrufende Task läuft also weiter). Eine laufende Task kann eine andere laufende Task auch stoppen, indem sie den **`stop`** - Befehl verwendet. Später kann diese Task wieder gestartet werden, sie beginnt allerdings wieder am Anfang und nicht an der Stelle, an der sie zuvor gestoppt wurde.
Wir wollen uns die Verwendung von Tasks an einem Beispiel verdeutlichen. Dazu verwenden wir das Programm, welches den Roboter im Quadrat fahren ließ, mit dem Unterschied, dass unser Roboter nun auf ein Hindernis reagieren soll.
Es ist schwer dies mit einer Task zu programmieren, da der Roboter zwei Dinge gleichzeitig tun muß: Erstens im Quadrat fahren, das heißt die Motoren entsprechend zu steuern, und zweitens den Berührungssensor überwachen. Daher ist es besser zwei Tasks zu verwenden. Eine, die das Fahren im Quadrat steuert und eine zweite, welche den Sensor überwacht und bei einer Berührung das Ausweichmanöver übernimmt.
Ist hier das Programm:

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH); //definert den Sensor
  start check_sensors;              //startet Task, die Sensor überwacht
  start move_square;                //startet Task, die Quadrate fährt
}

task move_square()                  //Task für Quadrate
{
  while (true)                      //unsere Endlosschleife
  {
    OnFwd(OUT_A+OUT_C); Wait(200);  //fährt 2 Sekunde geradeaus
    OnRev(OUT_C); Wait(160);         //und für 1,6 Sekunden nach rechts
  }
}

task check_sensors()                //Task für Sensor
{
  while (true)                      //ebenfalls eine Endlosschleife
  {
    if (SENSOR_1 == 1)              //fragt ab, ob Sensor 1 betätigt
    {
      stop move_square;             //stoppt Task für Quadrat
      OnRev(OUT_A+OUT_C); Wait(50); //fährt 0,5 Sekunden rückwärts
      OnFwd(OUT_A); Wait(80);       //startet Ausweichmanöver
      start move_square;            //startet Task für Quadrat
    }
  }
}
```

Die **`task`** `main()` teilt dem RCX nur den Sensortyp mit und startet die beiden anderen Tasks. Danach wird die **`task`** `main ()` beendet. Die task `move_square` (move = bewegen, square = Quadrat) veranlaßt den Roboter ständig im Quadrat zu fahren. Die task `check_sensors` (check = überprüfen,) überprüft, ob der Berührungs- Sensor gedrückt ist. Ist das der Fall, so passiert folgendes: zunächst wird die task `move_square` gestoppt. Dieses ist sehr wichtig, denn `check_sensors` übernimmt jetzt Steuerung über die Bewegungen des Roboters und fährt den Roboter zunächst ein Stück zurück. Dann führt sie eine Drehung aus und startet die Task `move_square` wieder.

Weshalb ist es nun so wichtig, dass die task `check_sensors` die task `move_square` unterbricht?
Nun, wie wir ja schon gelernt haben, laufen beide Tasks gleichzeitig. Würden nun beide Tasks unterschiedliche Befehle an denselben Motor geben, wüßte der RCX nicht, was er nun tun soll, und unser Roboter würde sich komisch verhalten.
Im Kapitel 10 gehen wir auf dieses Problem noch einmal näher ein.

7.2. Unterprogramme

Manchmal benötigt man exakt dieselbe Abfolge von Programmschritten an mehreren Stellen eines Programms. Man kann sie nun mehrmals schreiben oder einfach als Unterprogramm von der gewünschten Stelle aus aufrufen. Im RCX können wir bis zu 8 Unterprogramme verwenden. Schauen wir uns dies an einem Beispiel an:

```
sub turn_around()          //definiert das Unterprogramm turn_around
(),
{                          //welches das Ausweichmanöver durchführen
  OnRev(OUT_C); Wait(340); //soll
  OnFwd(OUT_A+OUT_C);
}

task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(100);
  turn_around();          //von hier aus wird turn_around() aufgerufen
  Wait(200);
  turn_around();
  Wait(100);
  turn_around();
  Off(OUT_A+OUT_C);
}
```

In diesem Programm haben wir ein Unterprogramm definiert, welches den Roboter auf der Stelle drehen läßt, indem der Motor C in den Rückwärtsgang geschaltet wird, während sich der Motor A noch vorwärts dreht. Die **`task`** `main()` ruft das Unterprogramm `turn_arround()` (turn = drehen, arround = herum) dreimal auf. Beachte bitte, dass das Unterprogramm einfach durch seinen Namen aufgerufen wird. Dabei ist wichtig, dass du die Klammern ( ) hinter dem Namen nicht vergißt. Der Aufbau der Bezeichnung für Unterprogramme ist also derselbe, wie der von Anweisungen, nur dass die Klammer ( ) leer bleibt.

Achtung!:

- Unterprogramme verursachen häufig Probleme. Der Vorteil ist, dass sie nur einmal im RCX gespeichert werden, was dort natürlich Speicherplatz spart.
- Unterprogramme können nicht von anderen Unterprogrammen aus aufgerufen werden.
- Unterprogramme können von verschiedenen Tasks aufgerufenen werden. Das ist aber nicht ratsam, weil dann dasselbe Unterprogramm unter Umständen gleichzeitig von mehreren Tasks aufgerufen wird, was zu unvorhersehbaren Reaktionen des Roboters führt!

7.3. Inline- Funktionen

Wie bereits oben erwähnt, verursachen Unterprogramme manchmal Probleme. Der Vorteil der Unterprogramme liegt darin, dass sie nur einmal im RCX gespeichert werden. Dieses spart Speicherplatz im RCX. Aber wenn du nur kurze Unterprogramme hast, verwende statt dessen besser die Inline – Funktion. Damit wird der Programmcode an alle die Stellen kopiert, von denen du die Funktion aus aufrufst. Dieses erfordert zwar mehr Speicherplatz, aber Probleme wie bei Unterprogrammen kommen damit nicht vor. Auch gibt es keine Begrenzung für die Zahl der Inline– Funktionen.
Definieren und Benennen von Inline- Funktionen geht genauso, wie bei Unterprogrammen. Verwende nur das Schlüsselwort **`void`** statt **`sub`**. (**`void`** wurde verwendet, da es auch in anderen Programmiersprachen, wie C, eingesetzt wird.)
So schaut das oben genannte Beispiel mit Inline- Funktionen aus:

```
void turn_around()         // definert Inline-Funktion
{
  OnRev(OUT_C); Wait(340);
  OnFwd(OUT_A+OUT_C);
}
task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(100);
  turn_around();           // ruft die Inline-Funktion auf
  Wait(200);
  turn_around();           // ruft die Inline-Funktion auf
  Wait(100);
  turn_around();           // ruft die Inline-Funktion auf
  Off(OUT_A+OUT_C);
}
```

Inline- Funktionen haben einen weiteren Vorteil gegenüber Unterprogrammen. Sie können Argumente haben.
Ein Argument ist der Wert in der Klammer hinter der Inline– Funktion. Dieser Wert wird beim Aufruf mit übergeben. Du kannst damit den Ablauf der Funktion von der Stelle aus, von der du sie aufrufst, beeinflussen.
Schauen wir uns das doch an einem Beispiel an:

```
void turn_around(int turntime)  //definiert Inline- Funktion mit der
{                               //Variablen turntime
  OnRev(OUT_C); Wait(turntime); //Der Roboter soll sich turntime drehen
  OnFwd(OUT_A+OUT_C);
}

task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(100);
  turn_around(200);             //ruft turn_around auf und übergibt dabei
  Wait(200);                    //den Wert 200 für die Drehzeit
  turn_around(50);              //ruft turn_around auf und übergibt dabei
  Wait(100);                    //den Wert 100 für die Drehzeit
  turn_around(300);             //ruft turn_around auf und übergibt dabei
  Off(OUT_A+OUT_C);             //den Wert 300 für die Drehzeit
}
```

Beachte, dass in der Klammer hinter dem Namen der Inline- Funktion die Variable `turntime` als ganzzahlig (**`int`** = integer) definiert wird.
Inline-Funktionen können auch mehrere Argumente haben, die jede für sich durch Komma getrennt, definiert wird.
Das nächste Beispiel zeigt dies: **`void`** `turn_around(`**`int`** `turntime, `**`int`** `Power)`
Der Befehl `SetPower()` stellt die Ausgangsleistung der Motoren ein (Power = Leistung, siehe Kapitel 1.5 und 8).

```
void turn_around(int turntime, int Power)  //Inline- Funktion mit der
{                                  //Variablen turntime und Power
  SetPower (OUT_A+OUT_C,Power);  //Stellt die Motorleistung ein
  OnRev(OUT_C); Wait(turntime);  //Der Roboter soll sich turntime
drehen
  OnFwd(OUT_A+OUT_C);
}

task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(100);
  turn_around(200,3);          //ruft turn_around auf und übergibt dabei
  Wait(200);                   //die Werte 200 und 3
  turn_around(50,5);           //ruft turn_around auf und übergibt dabei
  Wait(100);                   // die Werte 50 und 5
  turn_around(300,7);          //ruft turn_around auf und übergibt dabei
  Off(OUT_A+OUT_C);            // die Werte 300 und 7
}
```

7.4. Die Definition von Makros

Es gibt noch eine andere Methode einen Programmabschnitt mit einem eigenen Namen zu versehen: Makros.
Verwechsle dies aber bitte nicht mit den Makros im **RCX-Command Center**.
Wir haben zuvor gesehen, dass wir Konstanten mit `#define` definieren können, indem wir ihnen einen Namen geben.
Genauso können wir einen Programmabschnitt definieren. Hier das gleiche Programm wieder, aber diesmal verwenden wir Makros:

```
#define turn_around OnRev(OUT_C);Wait(340);OnFwd(OUT_A+OUT_C);

task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(100);
  turn_around;
  Wait(200);
  turn_around;
  Wait(100);
  turn_around;
  Off(OUT_A+OUT_C);
}
```

Nach `#define` geben wir dem Makro seinen Namen, hier `turn_around` (= herumdrehen) und danach schreiben wir die Befehle, durch Strich- Punkte getrennt, alle in eine Zeile. Beachte, dass die Befehle in einer Zeile sein sollten.
Es gibt Möglichkeiten diese auf mehrere Zeilen zu verteilen, dies wird aber nicht empfohlen.
Die `#define` - Anweisung ist sehr leistungsfähig. Sie kann auch Argumente haben.
In unserem Beispiel verwenden wir vier Makros: je eines für rechts, links, vorwärts und rückwärts und jedes hat zwei Argumente: die Geschwindigkeit **`s`** und die Zeit **`t`**.

```
#define turn_right(s,t)  SetPower(OUT_A+OUT_C,s);OnFwd(OUT_A);OnRev(OUT_C);Wait(t);
#define turn_left(s,t)   SetPower(OUT_A+OUT_C,s);OnRev(OUT_A);OnFwd(OUT_C);Wait(t);
#define forwards(s,t)    SetPower(OUT_A+OUT_C,s);OnFwd(OUT_A+OUT_C);Wait(t);
#define backwards(s,t)   SetPower(OUT_A+OUT_C,s);OnRev(OUT_A+OUT_C);Wait(t);

task main()
{
  forwards(3,200);   //ruft Makro forwards (=vorwärts) mit den Werten 3 und 200 auf
  turn_left(7,160);  //ruft Makro turn_left (=drehe links) mit den Werten 7 und 160 auf
  forwards(7,100);   //ruft Makro forwards (=vorwärts) mit den Werten 7 und 100 auf
  backwards(7,200);  //ruft Makro backwards (=rückwärts) mit den Werten 7 und 200 auf
  forwards(7,100);   //ruft Makro forwards (=vorwärts) mit den Werten 7 und 100 auf
  turn_right(7,160); //ruft Makro turn_right (=drehe rechts) mit den Werten 7 und 160 auf
  forwards(3,200);   //ruft Makro forwards (=vorwärts) mit den Werten 3 und 200 auf
  Off(OUT_A+OUT_C);
}
```

Makros sind auch deshalb vorteilhaft, weil sie das Programm klein und damit übersichtlicher machen. Du hast es damit bei Änderungen leichter, wenn du zum Beispiel die Anschlüsse der Motoren änderst, dann genügt es die Makros zu verändern. Der Rest deines Programmes bleibt.

7.5. Zusammenfassung

In diesem Kapitel lernten wir die Verwendung von Aufgaben (Tasks), Unterprogrammen, Inline- Funktionen und Makros. Je nach Anwendung hat jedes seine Vor- und Nachteile.

Tasks können gleichzeitig ablaufen und nebeneinander die unterschiedlichsten Sachen erledigen.

Unterprogramme haben den Vorteil, dass derselbe Programmablauf von verschiedenen Stellen aus aufgerufen werden kann, aber nur einmal Speicherplatz belegt. Dies ist aber auch zugleich ihr größter Nachteil: Wenn das selbe Unterprogramm gleichzeitig von verschiedenen Stellen (insbesondere Tasks, die ja nebeneinander ablaufen) aufgerufen wird, gibt es Probleme.

Inline- Funktionen funktionieren im Prinzip wie Unterprogramme, du brauchst die Befehle nur einmal einzugeben. Intern im RCX wird diese Befehlsabfolge an all die Stellen kopiert, von wo aus die Inline- Funktion aufgerufen wird. Dadurch können sie, im Gegensatz zu Unterprogrammen, parallel ablaufen. Sie benötigen deshalb aber auch mehr Speicherplatz im RCX.
Im Gegensatz zu Unterprogrammen kann man Inline-Funktionen auch ein (oder mehrere) Argument(e) (= Wert(e)) mit übergeben.

Schließlich haben wir Makros kennengelernt. Sie eignen sich insbesondere für kurze Programmabschnitte, die von verschiedenen Stellen aus aufgerufen werden sollen. Da den Makros beim Aufruf auch mehrere Werte mit übergeben werden können, sind sie für die Programmierung besonders interessant.

Bravo !!

Du hast dich nun bis hierher durchgearbeitet und damit die Grundlagen für die Programmierung mit NQC gelernt.
Die anderen Kapitel dieser Anleitung sind für spezielle Anwendungen gedacht. Du mußt sie also nicht sofort lesen. Aber blättere ruhig einmal weiter. Sicher findest du interessante Anregungen für Deine LEGO- Modelle!

8. Der RCX macht Musik

Der RCX hat einen eingebauten Lautsprecher, mit dem er Klänge und sogar einfache Musikstücke spielen kann. Damit kannst du den RCX so programmieren, dass er etwas "sagen" kann. Aber es ist auch ganz lustig, wenn dein Roboter Musik spielt, während er herumfährt.

8.1. Vorprogrammierte Klänge

Im RCX sind 6 Klänge, die von 0 bis 5 nummeriert sind, vorprogrammiert. Sie klingen wie folgt.:

0 kurzes "Bip", das bei jedem Tastendruck ertönt.
1 "Piep Piep", das beim Einschalten des RCX erklingt.
2 Tonfolge von hoch nach tief
3 Tonfolge von tief nach hoch
4 'Buhhh' = Fehlerhinweis
5 schnelle Tonfolge von tief nach hoch, das bei erfolgreicher Übertragung eines Programms ertönt.

Du kannst sie mit dem Befehl `PlaySound()`abspielen. Hier ist ein kleines Programm, das alle Klänge nacheinander abspielt:.

```
task main()
{
  PlaySound(0); Wait(100);
  PlaySound(1); Wait(100);
  PlaySound(2); Wait(100);
  PlaySound(3); Wait(100);
  PlaySound(4); Wait(100);
  PlaySound(5); Wait(100);
}
```

Vielleicht wunderst du dich über die `Wait(100);`-Befehle. Der Grund dafür ist, dass der RCX beim `PlaySound( )`-Befehl nicht wartet, bis die Tonfolge abgespielt ist, sondern er führt sofort den folgenden Befehl durch. Der RCX hat einen kleinen Puffer, in dem er einige Töne speichern kann. Aber nach einer Weile ist dieser Puffer voll und Töne gehen verloren. Dieses ist bei Tönen nicht so schlimm, aber für Musik ist das sehr wichtig, wie wir weiter unten sehen werden.
Beachte bitte, das das Argument für `PlaySound( );` (also der Wert in Klammern) eine Konstante sein muß. Du darfst hier keine Variable einsetzen!

8.2. Musik

Für interessantere Musikstücke hat der NQC den Befehl `PlayTone(262,40);`
Er hat zwei Argumente (hier 262 und 40). Der erste Wert gibt die Tonfrequenz an (hier 262 Hertz), der zweite die Tondauer (hier 40/100 = 0,4 Sekunden).
Hier eine Tabelle mit nützlichen Frequenzen:

Sound	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
G#	52	104	208	415	831	1661	3322	
G	49	98	196	392	784	1568	3136	
F#	46	92	185	370	740	1480	2960	
F	44	87	175	349	698	1397	2794	
E	41	82	165	330	659	1319	2637	
D#	39	78	156	311	622	1245	2489	
D	37	73	147	294	587	1175	2349	
C#	35	69	139	277	554	1109	2217	
C	33	65	131	262	523	1047	2093	4186
B	31	62	123	247	494	988	1976	3951
A#	29	58	117	233	466	932	1865	3729
A	28	55	110	220	440	880	1760	3520

Was wir bereits oben für Töne gelernt haben, gilt auch hier. Der RCX wartet nicht, bis der Ton zu Ende gespielt ist. Wenn du also eine Tonfolge für ein Lied programmierst, füge besser `Wait();` -Befehle hinzu, die etwas länger sein sollten, als die Tondauer.

Hier ist ein Beispiel:

```
task main()
{
  PlayTone(262,40);  Wait(50); // der Wait()-Befehl ist 10/100 Sekunden
  PlayTone(294,40);  Wait(50); //länger als die Tondauer. Dies sorgt
  PlayTone(330,40);  Wait(50); //dafür, dass die Töne sauber
voneinander
  PlayTone(294,40);  Wait(50); //getrennt abgespielt werden.
  PlayTone(262,160); Wait(200);
}
```

Wenn du möchtest, dass der RCX Musik spielt während er herumfährt, dann programmiere die Musik in einer eigenen Task. Diese ist dann ausschließlich für die Musik zuständig und läuft unabhängig vom übrigen Programm.
Hier haben wir ein Beispiel eines ziemlich albernen Programms, mit dem der RCX hin und her fährt und andauernd eine Melodie spielt.

```
task music()                  //ausschließlich für die Musik zuständig
{
  while (true)
  {
    PlayTone(262,40);  Wait(50);
    PlayTone(294,40);  Wait(50);
    PlayTone(330,40);  Wait(50);
    PlayTone(294,40);  Wait(50);
  }
}

task main()
{
  start music;                //startet die Musik
  while(true)                 //läßt Roboter dauernd hin- und her fahren
  {
    OnFwd(OUT_A+OUT_C); Wait(300);
    OnRev(OUT_A+OUT_C); Wait(300);
  }
}
```

Du kannst die Musik auch mit dem “Klavier” spielen, das ein Teil des **RCX-Command Center** ist. (Tools, RCX- Piano) Spiele doch einfach ein Musikstückchen mit dem RCX- Piano. Achte dabei auf die Tondauer. Du kannst dein Musikstück über Save (engl. = speichern) abspeichern und anschließend im **RCX-Command Center** laden und bearbeiten, also beispielsweise die Tonlänge anpassen. Hier ist ein Beispiel, das ich so erstellt habe:

```
// Music file created by RCX Command Center.

#define __NOTETIME   10
#define __WAITTIME   12

task main()
{
  PlayTone(523,4*__NOTETIME); Wait(4*__WAITTIME); //Tonlänge und Dauer
  PlayTone(659,2*__NOTETIME); Wait(2*__WAITTIME); //der Programmunter-
  PlayTone(523,2*__NOTETIME); Wait(2*__WAITTIME); //brechung sind ein-
  PlayTone(392,4*__NOTETIME); Wait(4*__WAITTIME); //ander angepasst!
  PlayTone(392,4*__NOTETIME); Wait(4*__WAITTIME);
  PlayTone(392,2*__NOTETIME); Wait(2*__WAITTIME);
  PlayTone(523,2*__NOTETIME); Wait(2*__WAITTIME);
  PlayTone(392,2*__NOTETIME); Wait(2*__WAITTIME);
  PlayTone(330,2*__NOTETIME); Wait(2*__WAITTIME);
  PlayTone(262,4*__NOTETIME); Wait(4*__WAITTIME);
}
```

erkennst du die Melodie?

8.3. Zusammenfassung

In diesem Kapitel hast du gelernt, wie du deinem RCX Töne und Melodien beibringen kannst.
Du hast auch gelernt, dass es meist günstiger ist, wenn du die Melodie in einer eigenen Task programmierst.

9. Mehr über Motoren

Es gibt eine Anzahl weiterer Befehle, mit deren Hilfe du die Motoren exakter steuern kannst. In diesem Kapitel behandeln wir sie.

9.1. Motoren auslaufen lassen

Wenn du den Befehl `Off()` verwendest, stoppt der Motor sofort mit der Bremse. In NQC ist es auch möglich, die Motoren in einer sanfteren Weise zu stoppen und nicht die Bremse zu verwenden. Dafür verwenden wir den Befehl `Float()`.

Schauen wir uns dazu das nachfolgende Beispiel an: Zuerst stoppt der Roboter schnell, indem er die Bremsen verwendet, dann stoppt er langsam, weil nur der Strom abgeschaltet wird. Das ist so, wie wenn du mit deinem Fahrrad auf gerader Stecke aufhörst zu treten. Dein Fahrrad rollt dann auch noch ein Stück weiter.
Je nachdem, ob sich die Räder bei deinem Roboter leichter oder etwas schwerer Drehen, hält dieser nun schneller oder langsamer an.

```
task main()
{
  OnFwd(OUT_A+OUT_C);
  Wait(200);
  Off(OUT_A+OUT_C);    //stoppt mit Bremse
  Wait(100);
  OnFwd(OUT_A+OUT_C);
  Wait(200);
  Float(OUT_A+OUT_C);  //stoppt, indem die Motoren langsam auslaufen
}
```

Bei unserem Roverbot ist der Effekt natürlich nicht so groß, wie bei einem schnell laufenden Radfahrzeug.

Interessant ist in diesem Zusammenhang auch folgende Beobachtung: Wenn du die Motoren mit dem `Off ()`-Befehl gestoppt hast, lassen sie sich nur schwer drehen, so lange der RCX noch eingeschaltet ist. Hast du die Motoren dagegen mit dem `Float ()` Befehl gestoppt, so lassen sie sich leicht drehen.

9.2. Weiterentwickelte Befehle

Der Befehl `OnFwd()` erledigt in Wirklichkeit zwei Dinge: er stellt die Richtung auf vorwärts ein und schaltet den Motor an. Der Befehl `OnRev()` erledigt auch zwei Dinge: er stellt die Richtung auf rückwärts und schaltet den Motor ein.
NQC hat auch Befehle, um diese beiden Dinge getrennt zu tun. Wenn du nur den Motor einschalten oder die Richtung ändern möchtest, ist es leistungsfähiger, wenn du diese unterschiedlichen Befehle verwendest. Das benötigt weniger Speicher im RCX, ist es schneller, und es kann daher glattere Bewegungen ergeben.
Die zwei unterschiedlichen Befehle sind `SetDirection()`, der die Richtung einstellt, (`OUT_FWD`, `OUT_REV` oder `OUT_TOGGLE`, welcher die Drehrichtung hin- und her schaltet), und `SetOutput()`, welcher die Voreinstellung des Motors verändert (`OUT_ON`, `OUT_OFF` oder `OUT_FLOAT`).
Hier ist ein einfaches Programm, das den Roboter vorwärts, rückwärts und wieder vorwärts fahren läßt.

```
task main()
{
  SetPower(OUT_A+OUT_C,7);            //stellt Leistung des Motors
ein
  SetDirection(OUT_A+OUT_C,OUT_FWD);  //stellt Drehrichtung ein
  SetOutput(OUT_A+OUT_C,OUT_ON);      //schaltet Motoren ein
  Wait(200);
  SetDirection(OUT_A+OUT_C,OUT_REV);  //schaltet in den Rückwärtsgang
  Wait(200);
  SetDirection(OUT_A+OUT_C,OUT_TOGGLE); //ändert Drehrichtung
  Wait(200);
  SetOutput(OUT_A+OUT_C,OUT_FLOAT);    //schaltet Motoren ab
}
```

Beachte: beim Start jedes Programms werden alle Motoren in Vorwärtsrichtung und die Geschwindigkeit auf 7 eingestellt. So sind im oben genannten Beispiel die ersten beiden Befehle nicht notwendig.
Es gibt eine Anzahl weiterer Bewegungsbefehle, die Abkürzungen für Kombinationen der Befehle oben sind.

Hier ist eine komplette Liste:

`On('motors')`	schaltet die Motoren an
`Off('motors')`	schaltet die Motoren ab (mit Bremse)
`Float('motors')`	schaltet die Motoren ab (auslaufen lassen)
`Fwd('motors')`	schaltet die Motoren in den Vorwärtsgang, ohne sie einzuschalten
`Rev('motors')`	schaltet die Motoren in den Rückwärtsgang, ohne sie einzuschalten
`Toggle('motors')`	schaltet die Richtung hin- und her (vorwärts nach rückwärts und umgekehrt)
`OnFwd('motors')`	schaltet die Motoren in den Vorwärtsgang und schaltet sie ein
`OnRev('motors')`	schaltet die Motoren in den Rückwärtsgang und schaltet sie ein
`OnFor('motors','ticks')`	schaltet den Motor für eine gewisse Zeit (ticks/100 Sekunden) ein
`SetOutput('motors','mode')`	bestimmt die Einstellung der Ausgänge (`OUT_ON`, `OUT_OFF` oder `OUT_FLOAT`)
`SetDirection('motors','dir')`	bestimmt die Drehrichtung (`OUT_FWD`, `OUT_REV` oder `OUT_TOGGLE`)
`SetPower('motors','power')`	stellt die Leistung der Motoren ein (0-9)

`'motors'` steht für `OUT_A`, `OUT_B`, `OUT_C` oder eine Kombination, wie `OUT_A+OUT_C`

9.3. Unterschiedliche Geschwindigkeiten

Vermutlich hast du schon bemerkt, dass eine Änderung der Motorgeschwindigkeit im Programm am ROVERBOT nur eine geringe Auswirkung hat. Das liegt daran, dass eigentlich nicht die Geschwindigkeit, sondern die Drehkraft (der Fachmann spricht hier von Drehmoment) geändert wird. Daher wird der Unterschied erst richtig sichtbar, wenn dein Motor so richtig arbeiten muß, und dann noch ist der Unterschied zwischen 2 und 7 eher klein.
Um die Geschwindigkeit als solche genauer zu bestimmen, ist es besser die Motoren in schneller Abfolge ein- und wieder auszuschalten.. Hier ist ein einfaches Programm, das dies tut. Es hat eine **`task`** `run_motor()` , welche die Motoren steuert. Es überprüft ständig die Variable `speed` (engl. = Geschwindigkeit), um zu sehen, wie hoch die aktuelle Geschwindigkeit ist. Positiv ist vorwärts, negativ rückwärts. Es stellt die Motoren in die entsprechende Richtung ein und wartet dann, je nach Geschwindigkeit einige Zeit, bevor es die Motoren wieder ausschaltet.
Die Haupttask stellt nur Geschwindigkeiten und Wartezeiten ein, startet und stoppt die **`task`** `run_motor()` .

```
int speed, __speed;

task run_motor()
{
  while (true)
  {
    __speed = speed;
    if (__speed > 0) {OnFwd(OUT_A+OUT_C);}
    if (__speed < 0) {OnRev(OUT_A+OUT_C); __speed = -__speed;}
    Wait(__speed);
    Off(OUT_A+OUT_C);
  }
}

task main()
{
  speed = 0;
  start run_motor;           //startet die Task run_motor()
  speed = 1;   Wait(200);    //da run_motor() noch läuft, hat eine
  speed = -10; Wait(200);    //Veränderung der Variablen Einfluß
  speed = 5;   Wait(200);    //auf das Verhalten der Motoren
  speed = -2;  Wait(200);
  stop run_motor;
  Off(OUT_A+OUT_C);
}
```

Dieses Programm kann natürlich sehr viel leistungsfähiger gestaltet werden, indem du Drehungen programmierst oder einen `Wait()` –Befehl nach dem Befehl `Off()` –Befehl einfügst. Probiere es einfach einmal.

9.4. Zusammenfassung

In diesem Kapitel haben wir die erweiterten Motoren- Befehle gelernt: `Float()`, der den Motor langsam stoppt, `SetDirection()`, der die Richtung einstellt `OUT_FWD`, `OUT_REV` oder `OUT_TOGGLE`, welcher die gegenwärtige Drehrichtung umschaltet, und `SetOutput()`, der den Modus einstellt (`OUT_ON`, `OUT_OFF` oder `OUT_FLOAT`).
Wir kennen nun die komplette Liste aller Steuerbefehl für die Motoren.
Wir lernten auch einen Trick kennen, mit dessen Hilfe die Bewegungsgeschwindigkeit genauer gesteuert werden kann.

10.Mehr über Sensoren

In Kapitel 6 lernten wir die Grundlagen über den Einsatz von Sensoren. Mit Sensoren kannst du aber viel mehr tun. In diesem Kapitel behandeln wir den Unterschied zwischen Sensor- Modus und Sensor- Art. Wir werden lernen, wie man den Dreh- Sensor benutzt (ein Sensor, der nicht mit deinem LEGO- Mindstorms geliefert wurde, aber separat gekauft werden kann und sehr interessant ist), und wir werden einige Tricks kennen lernen, wie du mehr als drei Sensoren an deinen RCX anschließen kannst und, wie du einen Näherungssensor bauen kannst.

10.1. Sensor- Modus und Sensor- Art

Der `SetSensor()`- Befehl, den wir in Kapitel 6 kennen lernten, bewirkt in Wirklichkeit zwei Dinge: er stellt die Art des Sensors ein, und er stellt den Modus ein, in dem der Sensor funktioniert. Dadurch, dass du den Modus und die Art des Sensors getrennt einstellst, kannst du das Verhalten des Sensors genauer steuern, was für bestimmte Anwendungen nützlich ist.

Die Art, wie der Sensors vom RCX abgefragt wird, wird mit dem Befehl `SetSensorType()` eingestellt.
Es gibt vier unterschiedliche Arten: `SENSOR_TYPE_TOUCH`, das ist der Berührungs- Sensor, `SENSOR_type_LIGHT`, das ist der Lichtsensor, `SENSOR_TYPE_TEMPEATURE`, das ist der Temperaturfühler und `SENSOR_TYPE_ROTATION`, der Umdrehungs- Sensor. Temperatur- und Umdrehungs- Sensor sind nicht in deinem LEGO-Mindstorms- Kasten enthalten, können aber extra gekauft werden.
Über die Sensorart kannst du beispielsweise den Licht- Sensor abstellen, so dass dieser keine Energie verbraucht.

Mit dem Sensor- Modus wird festgelegt, wie der RCX die Sensor- Signal auswertet (interpretiert). Er wird mit dem Befehl `SetSensorMode()` eingestellt. Es gibt acht unterschiedliche Modi.

SENSOR_MODE_RAW

Der wichtigste ist `SENSOR_MODE_RAW`. In diesem Modus beträgt der Wert beim Abfragen des Sensors zwischen 0 und 1023. Es ist der ursprüngliche (rohe) Wert, der durch den Sensor erzeugt wird. Was dieser Wert bedeutet, hängt vom tatsächlichen Sensor ab.
Beispielsweise ergibt sich ein Wert nahe bei 1023, wenn der Berührungssensor nicht gedrückt wird. Wird er dagegen fest gedrückt, ist er nahe bei 50. Wenn er teilweise gedrückt wird, erstreckt sich der Wert zwischen 50 und 1000.
Wenn du also einen Berührungssensor auf RAW- Modus einstellst, kannst du tatsächlich herausfinden, ob er teilweise betätigt wird.
Wenn der Sensor ein Licht- Sensor ist, reicht der Wert von ungefähr 300 (sehr hell) bis 800 (sehr dunkel). Dieses gibt einen viel genaueren Wert, als mit dem `SetSensor()` Befehl.

SENSOR_MODE_BOOL

Der zweite Sensor- Modus ist `SENSOR_MODE_BOOL`. In diesem Modus ist der Wert genau 0 oder 1. Wenn der rohe Wert über ungefähr 550 ist, ist der Wert 0, andernfalls ist er 1. `SENSOR_MODE_BOOL` ist die Standardeinstellung für einen Berührungssensor.

SENSOR_MODE_PERCENT

`SENSOR_MODE_PERCENT` wandelt den rohen Wert in einen Wert zwischen 0 und 100. Jeder rohe Wert von 400 oder niedriger wird zu 100 Prozent. Wenn der rohe Wert steigt, geht der Prozentsatz langsam nach unten bis 0.
`SENSOR_MODE_PERCENT` ist die Standardeinstellung für einen Licht- Sensor.

SENSOR_MODE_EDGE und SENSOR_MODE_PULSE

Darüber hinaus gibt es zwei andere interessante Modi: `SENSOR_MODE_EDGE` und `SENSOR_MODE_PULSE`. Sie zählen Übergänge. Das sind Änderungen von einem tiefen zu einem hohen rohen Wert oder umgekehrt. Wenn du beispielsweise einen Berührungssensor betätigst, verursacht dieses einen Übergang von einem hohen zu einem niedrigen rohen Wert. Wenn du ihn losläßt, ist das ein Übergang in die andere Richtung. Wenn du den Sensor- Modus auf `SENSOR_MODE_PULSE` einstellst, werden nur Übergänge von Tief nach Hoch gezählt. Auf diese Art wird jedes Berühren und Wiederloslassen als ein Ereignis gezählt.
Wenn du den Sensor- Modus auf `SENSOR_MODE_EDGE` einstellst, werden beide Übergänge gezählt, jedes Berühren und Wiederloslassen zählt doppelt. Damit kannst du ermitteln, wie oft ein Berührungssensor betätigt wurde, oder du kannst ihn in Verbindung mit einem Licht- Sensor verwenden, um zu zählen, wie oft eine (starke) Lampe an und ausgeschaltet wurde.

ClearSensor()

Wenn du Ereignisse zählst, solltest du selbstverständlich den Zähler zurück auf 0 stellen können. Dafür verwendest du den Befehl `ClearSensor()`. Er löscht den Zähler für den entsprechenden Sensor.
Wir wollen uns das an einem Beispiel anschauen. Das folgende Programm benutzt einen Berührungs- Sensor, um den Roboter zu steuern. Schließe den Berührungs- Sensor mit einer langen Leitung an Eingang 1 an. Wenn du den Berührungs-Sensor schnell zweimal hintereinander drückst, fährt der Roboter vorwärts, wenn du ihn nur einmal drückst, hält er an.

```
task main()
{
  SetSensorType(SENSOR_1,SENSOR_TYPE_TOUCH);
  SetSensorMode(SENSOR_1,SENSOR_MODE_PULSE);
  while(true)
  {
    ClearSensor(SENSOR_1);
    until (SENSOR_1 >0);
    Wait(100);
    if (SENSOR_1 == 1) {Off(OUT_A+OUT_C);}
    if (SENSOR_1 == 2) {OnFwd(OUT_A+OUT_C);}
  }
}
```

Probiere dann das Programm noch mit `SENSOR_MODE_EDGE` statt `SENSOR_MODE_PULSE`. Was hat sich verändert?

Beachte, dass wir zuerst die Art des Sensors und dann des Modus einstellen. Das ist wichtig, weil das Ändern der Art des Sensors auch den Modus ändert.
`SENSOR_MODE_ROTATION` ist nur für einen Umdrehungs- Sensor sinnvoll (siehe unten).
Die Modi `SENSOR_mode_CELSIUS` und `SENSOR_MODE_FAHRENHEIT` sind mit nur Temperaturfühlern sinnvoll und liefern die Temperatur in der angezeigten Weise.

10.2. Der Umdrehungs- Sensor

Der Umdrehungs- Sensor ist sehr nützlich. Leider ist er nicht im Standard- Baukasten enthalten. Er kann aber extra von LEGO gekauft werden. Der Umdrehungs- Sensor enthält eine Bohrung, durch die eine Welle hindurch geführt werden kann. Er ermittelt wie weit die Welle gedreht wird. Eine volle Umdrehung der Welle ergibt 16 Schritte (oder -16, wenn du ihn in die andere Richtung drehst). Umdrehungs- Sensoren kann man dazu verwenden den Roboter genau kontrollierte Bewegungen ausführen zu lassen. Du kannst damit genau bestimmen, wie weit sich eine Welle drehen soll. Wenn du eine feinere Auflösung als 16 Schritte benötigst, kannst du die Drehzahl des Sensors über ein Zahnradgetriebe erhöhen.

Eine Standardanwendung ist folgende: Unser Roboter soll zwei Umdrehungs- Sensoren haben, die an den beiden Rädern angeschlossen werden, mit denen du ihn steuerst. Für eine Geradeausfahren müssen sich beide Räder gleichmäßig schnell drehen. Leider laufen die Motoren normalerweise nicht mit genau der gleichen Geschwindigkeit. Mit den Umdrehungs- Sensoren kannst du feststellen, ob sich ein Rad schneller dreht. Du kannst diesen Motor (am besten mit `Float()`) vorübergehend stoppen, bis beide Motoren dieselbe Drehzahl haben. Das folgende Programm tut dies. Es läßt einfach den Roboter geradeaus fahren. Um es zu verwenden, baue zwei Umdrehungs- Sensoren an die beiden Räder und verbinde sie mit den Eingängen 1 und 3 des RCX.

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_ROTATION); ClearSensor(SENSOR_1);
  SetSensor(SENSOR_3,SENSOR_ROTATION); ClearSensor(SENSOR_3);
  while (true)
  {
    if (SENSOR_1 < SENSOR_3)          //rechtes Rad schneller
      {OnFwd(OUT_A); Float(OUT_C);}   //laß' rechtes Rad auslaufen
    else if (SENSOR_1 > SENSOR_3)     //linkes Rad schneller
      {OnFwd(OUT_C); Float(OUT_A);}   //laß linkes Rad auslaufen
    else                              //beide Räder gleich schnell
      {OnFwd(OUT_A+OUT_C);}           //schalte beide Räder ein
  }
}
```

Das Programm zeigt zuerst an, dass beide Sensoren Umdrehungs- Sensoren sind und setzt die Werte auf null zurück. Zunächst beginnt eine Endlosschleife (**while** (**true**)). In der Schleife überprüfen wir, ob die zwei Sensor- Messwerte gleich sind. Wenn das der Fall ist, fährt der Roboter geradeaus. Wenn einer größer ist, wird der entsprechende Motor gestoppt, bis beide Messwerte wieder gleich sind. Scheinbar ist dieses nur ein sehr einfaches Programm. Aber du kannst es so verändern, dass dein Roboter genaue Stecken zurücklegt oder dass dein Roboter exakte Kurven, beispielsweise im rechten Winkel, fährt.

10.3. Mehrere Sensoren an einem Eingang

Der RCX hat nur drei Eingänge, also können nur drei Sensoren an ihn angeschlossen werden. Wenn du kompliziertere Roboter bauen möchtest und einige Extra- Sensoren gekauft hast, reicht dies möglicherweise nicht aus. Glücklicherweise kannst du mit einigen Tricks zwei (oder sogar mehr) Sensoren an einen Eingang anschließen.

Der einfachste Fall ist, wenn du zwei Berührungssensoren an einen Eingang anschließt. Wenn einer von ihnen (oder beide) berührt wird, ist der Wert 1, andernfalls ist er 0. Du kannst die beiden zwar nicht unterscheiden, aber manchmal ist dieses

nicht notwendig. Wenn du zum Beispiel einen Berührungssensor an die Vorderseite und einen an die Rückseite des Roboters setzt, weißt du, dass derjenige, welcher betätigt wurde, nur der in Fahrtrichtung liegende sein kann. Aber du kannst den Modus des Eingangs auch auf “roh” einstellen (siehe oben). Jetzt erhältst du viel genauere Informationen. Wenn du Glück hast, haben die beiden Sensoren beim Betätigen unterschiedliche rohe Werte. Ist dieses der Fall, so kannst du zwischen den beiden Sensoren wirklich unterscheiden. Und wenn beide gleichzeitig betätigt werden, erhältst du einen viel niedrigeren Wert (etwa 30) als sonst, und damit kannst du auch dieses feststellen.

Du kannst auch einen Berührungs- und einen Licht- Sensor an einen Eingang anschließen. Stelle dann den Eingang auf `SENSOR_MODE_LIGHT` (sonst funktioniert der Lichtsensor nicht) und den Modus auf “roh” ein. Wenn der Berührungssensor betätigt wird, erhältst du in diesem Fall einen rohen Wert unter 100. Wenn er nicht gedrückt wird, erhältst du den Wert des Licht- Sensors, der nie unter 100 ist. Das folgende Programm verwendet diese Idee. Der Roboter muß mit einem Licht- Sensor, der nach unten zeigt und einem Stoßfänger, der mit einem Berührungssensor verbunden ist, ausgerüstet werden. Schließe beide an Eingang 1 an. Der Roboter bewegt sich ziellos innerhalb der Testbahn auf deinem Poster. Wenn der Licht- Sensor auf die dunkle Bahn trifft (roher Wert > 750), fährt der Roboter etwas zurück. Wenn der Roboter gegen ein Hindernis fährt (roher Wert unter 100), tut er dasselbe. Ist hier das Programm:

```
int ttt,tt2;

task moverandom()
{
  while (true)
  {
    ttt = Random(50) + 40;
    tt2 = Random(1);
    if (tt2 > 0)
      { OnRev(OUT_A); OnFwd(OUT_C); Wait(ttt); }
    else
      { OnRev(OUT_C); OnFwd(OUT_A);Wait(ttt); }
    ttt = Random(150) + 50;
    OnFwd(OUT_A+OUT_C);Wait(ttt);
  }
}

task main()
{
  start moverandom;
  SetSensorType(SENSOR_1,SENSOR_TYPE_LIGHT);
  SetSensorMode(SENSOR_1,SENSOR_MODE_RAW);
  while (true)
  {
    if ((SENSOR_1 < 100) || (SENSOR_1 > 750)) // || bedeutet “oder”, also
    {                                         // wenn Taster oder
Lichtsensor
      stop moverandom;
      OnRev(OUT_A+OUT_C);Wait(30);
      start moverandom;
    }
  }
}
```

Hoffentlich kommst du mit diesem Programm klar. Es hat zwei Tasks. Die **`task`** `moverandom` läßt den Roboter willkürlich umherfahren (move = Bewegen, random = zufällig). Die **`task`** `main` startet zuerst `moverandom`, stellt den Sensor ein und wartet bis ein Sensorsignal auftritt. Wenn der Sensorwert zu nieder (berühren) oder zu hoch (aus dem weißen Bereich heraus) wird, stoppt es die willkürlichen Bewegungen, schaltet kurz den Rückwärtsgang ein und startet wieder die **`task`** `moverandom`.
Es ist auch möglich, zwei Licht- Sensoren an den gleichen Eingang anzuschließen. Auf diese Weise hängt der rohe Wert von der gesamten Lichtmenge beider Sensoren zusammen ab. Aber dieses ist ziemlich unklar und daher ziemlich schwierig in der Anwendung. Dagegen ist die Kombination mit einem anderen Sensortyp, wie Drehsensor oder Temperatursensor, durchaus sinnvoll und nützlich.

10.4. Ein Annäherungssensor

Mit den Berührungs- Sensoren kann dein Roboter reagieren, wenn er gegen etwas stößt. Aber es wäre viel schöner, wenn der Roboter reagieren könnte, bevor er gegen etwas stößt. Er sollte erkennen, dass er irgendeinem Hindernis zu nahe ist. Leider gibt es dafür keine Sensoren. Aber es gibt einen Trick, den wir dafür verwenden können.
Der Roboter hat einen Infrarotanschluß (Infrarot- Schnittstelle), mit dessen Hilfe er mit deinem Computer oder mit anderen Robotern Verbindung aufnehmen kann (mehr über die Kommunikation zwischen Robotern sehen wir in Kapitel 11).
Es fällt auf, dass der Licht- Sensor für Infrarotlicht sehr empfindlich ist. Damit können wir einen Annäherungssensor bauen. Dazu folgende Überlegungen: Eine Task sendet ständig ein Infrarot- Signal aus. Eine andere Task mißt die Schwankungen in der Lichthelligkeit, die von Hindernissen reflektiert wird. Je stärker sich die Helligkeit des reflektierten Lichts verändert, desto näher befindet sich der Roboter an einem Hindernis.

Um diese Überlegung zu verwirklichen, baue den Licht- Sensor auf deinen Roboter oberhalb der Infrarotschnittstelle. Auf diese Art mißt er nur reflektiertes Infrarotlicht. Schließe ihn an Eingang 2 an. Wir verwenden rohen Modus für den Licht-Sensor, um die Veränderungen in der Helligkeit besser zu unterscheiden. Hier ist ein einfaches Programm, das den Roboter geradeaus fahren läßt, bis er einem Hindernis zu nahe kommt und dann eine 90-Grad-Wendung nach rechts macht.

```
int lastlevel;           // um den letzten Wert zu speichern

task send_signal()       // soll IR-Signale aussenden
{
  while(true)            // unsere Endlosschleife
    {SendMessage(0); Wait(10);} //10 Signale pro Sekunde (alle 10/100S)
}

task check_signal()
{
  while(true)             // unsere Endlosschleife
  {
    lastlevel = SENSOR_2;          //speichert das aktuelle Sensorsignal
    if(SENSOR_2 > lastlevel + 200) //vergleicht, ob nun mehr Licht kommt
      {OnRev(OUT_C); Wait(160); OnFwd(OUT_A+OUT_C);} //Drehung nach rechts
  }
}

task main()
{
  SetSensorType(SENSOR_2, SENSOR_TYPE_LIGHT);
  SetSensorMode(SENSOR_2, SENSOR_MODE_RAW);
  OnFwd(OUT_A+OUT_C);
  start send_signal;
  start check_signal;
}
```

Das **`task`** `send_signal()` sendet 10 IR Signale mit dem Befehl `SendMessage(0)` pro Sekunde aus. Die **`task`** `check_signal()` erfaßt in einer Endlosschleife (**`while`**`(`**`true`**`)`) ständig den Wert des Licht- Sensors und überprüft, ob dieser gegenüber dem zuvor empfangenen Wert um mehr als 200 angestiegen ist. Dieses ist dies der Fall, wenn sich der Roboter einem Hindernis nähert, er macht dann eine 90-Grad- Wendung nach rechts und fährt geradeaus weiter. Der Wert von 200 ist ziemlich willkürlich. Wenn du ihn kleiner wählst, dreht sich der Roboter weiter entfernt vor dem Hindernis. Wenn du ihn vergrößerst, fährt der Roboter näher an ein Hindernis heran. Das hängt auch von der Art des Materials (insbesondere der Farbe) des Hindernises und der Helligkeit im Raum ab. Probiere es aus, oder überlege dir ein verbessertes Programm, um den richtigen zu Wert ermitteln.

Ein Nachteil dieser Methode ist, dass sie nur in einer Richtung funktioniert. Vermutlich wirst du zusätzliche Berührungs-Sensoren an den Seiten deines Roboters benötigen, um Zusammenstöße dort zu vermeiden. Aber die Methode ist für Roboter interessant, die sich in einem Labyrinth bewegen müssen. Ein anderer Nachteil ist, dass damit die Verbindung zwischen Computer und Roboter blockiert ist, weil die durch den Roboter ausgesandten Infrarot- Befehle eine Verbindung behindern. Es kann auch sein, dass dieser Roboter die Fernbedienung deines Fernsehers stört.

10.5. Zusammenfassung

In diesem Kapitel haben wir zusätzlichen Sensoren kennengelernt. Wir sahen, wie man die Art und den Modus eines Sensors getrennt einstellt und wie dieses verwendet werden könnte, um mehr Informationen aus einem Sensorsignal zu erhalten. Wir lernten, wie man den Dreh- Sensor benutzt, und wir sahen, wie man mehrere Sensoren gleichzeitig an einen Eingang des RCX anschließt.
Schließlich lernten wir einen Trick, um den Infrarotanschluß des Roboters in Verbindung mit einem Licht- Sensor als Annäherungssensor zu verwenden. Alle diese Tricks sind sehr nützlich, da Sensoren bei komplizierten Robotern eine große Rolle spielen.

11.Parallele Tasks (Aufgaben)

Wie wir bereits gelernt haben, können Task in NQC gleichzeitig oder wie man auch sagt, parallel ablaufen. Dieses ist eine tolle Eigenschaft. Es ermöglicht uns in einer Task Sensoren zu überwachen, während eine andere Task den Roboter steuert, und eine weitere Task Musik spielt. Aber parallele Tasks können auch Probleme verursachen. Eine Task kann andere beeinflussen. Im folgenden wollen wir ein fehlerhaftes Programm betrachten. Hier steuert eine Task den Roboter im Quadrat, so wie wir es schon so oft zuvor getan haben, und die zweite (`task check_sensors()`) überwacht den Sensor. Wenn der Sensor berührt wird, fährt der Roboter etwas zurück und macht eine 90-Grad-Wendung.

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH); //stellt Sensor ein
  start check_sensors;
  while (true)                      //fahre Quadrate
  {
    OnFwd(OUT_A+OUT_C); Wait(200);
    OnRev(OUT_C); Wait(85);
  }
}

task check_sensors()                //überwacht Sensor
{
  while (true)  //Endlosschleife, damit Sensor dauernd überwacht wird
  {
    if (SENSOR_1 == 1)              //wenn Sensor betätigt
    {
      OnRev(OUT_A+OUT_C);           //fahre zurück
      Wait(50);
      OnFwd(OUT_A);                 //drehe nach rechts
      Wait(160);
      OnFwd(OUT_C);                 //und fahre wieder geradeaus
    }
  }
}
```

Dieses sieht vermutlich wie ein tadellos gültiges Programm aus, und der Kompiler (=Übersetzungsprogramm) wandelt es auch ohne zu meckern um und überträgt das Programm auf den RCX. Wenn du es aber ausführst, wird dir wahrscheinlich ein etwas unerwartetes Verhalten auffallen.
Versuche folgendes: Laß‘ deinen Roboter ein Hindernis berühren, während er sich dreht. (Du kannst ja dazu den Berührungssensor mit der Hand auslösen.) Daraufhin fährt er kurz rückwärts, aber sofort wieder vorwärts und stößt gegen das Hindernis. Der Grund dafür ist, dass die Tasks sich gegenseitig behindern können.
Folgendes geschieht: Während sich der Roboter dreht, führt die Task den Befehl `Wait(85)` aus und stoppt damit für 85/100 Sekunden den Programmablauf. In diesem Moment kommt ein Signal vom Berührungs- Sensor. Dadurch tritt die Bedingung `if (SENSOR_1 == 1)` ein und schaltet den Roboter in den Rückwärtsgang. In diesem Moment ist die `Wait(85)` –Anweisung der Haupttask zu Ende, der Roboter erhält den Befehl geradeaus zu fahren und stößt gegen das Hindernis. Die zweite Task hat in diesem Moment den `Wait(160)-` Befehl und bemerkt daher den Zusammenstoß nicht. Dieses ist sicherlich nicht das Verhalten, das wir gerne hätten.
Das Problem liegt darin, dass die erste Task weiter läuft, während die zweite das Ausweichmanöver steuert. Beide Tasks steuern gleichzeitig die Motoren an und überlagern sich dadurch in widersprüchlichen Anweisungen. Damit weiß unser Roboter nicht so recht, was er eigentlich tun soll.

11.1. Stoppen und erneutes Starten von Tasks

Eine Möglichkeit der Problemlösung ist sicherzustellen, dass immer nur eine Task den Roboter steuert. Das haben wir in Kapitel 7.1 getan. Wir wollen uns das Programm nochmals anschauen:

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  start check_sensors;
  start move_square;
}

task move_square()
{
  while (true)
  {
    OnFwd(OUT_A+OUT_C); Wait(100);
    OnRev(OUT_C); Wait(85);
  }
}

task check_sensors()
{
  while (true)
  {
    if (SENSOR_1 == 1)
    {
      stop move_square;
      OnRev(OUT_A+OUT_C); Wait(100);
      OnFwd(OUT_A); Wait(160);
      start move_square;
    }
  }
}
```

Der springende Punkt ist, dass die task `check_sensors()` die Motoren erst ansteuert, nachdem sie die task `move_square` gestoppt hat. So kann diese die Bewegung vom Hindernis weg nicht behindern. Sobald das Ausweichmanöver beendet ist, startet sie `move_square` wieder. Obwohl dieses für das oben genannte Problem eine gute Lösung ist, gibt es ein weiteres Problem. Wenn wir `move_square` wieder starten, beginnt es wieder am Anfang und nicht an der Stelle an der es zuvor unterbrochen wurde. Dieses ist für unsere kleinen Tasks ohne große Bedeutung. Aber oft entspricht dies nicht dem von uns gewünschten Verhalten. Oftmals wäre es besser eine Task dort zu stoppen, wo sie sich gerade befindet, und später genau an dieser Stelle wieder fortzusetzen. Leider ist dies nicht möglich.

11.2. Standardtechnik der Semaphore

Eine häufig verwendete Möglichkeit zur Lösung dieses Problems besteht darin, eine Variable sozusagen als Signal zu verwenden. Diese soll anzuzeigen, welche Task gerade die Steuerung der Motoren ausübt. Die anderen Tasks dürfen die Steuerung der Motoren so lange nicht aufnehmen, bis die erste Task mit der Variablen anzeigt, dass sie fertig ist und damit die Steuerung erlaubt. Solch eine Variable wird häufig ein Semaphor genannt. „**sem**“ soll solch ein Semaphor sein. Wir nehmen an, dass ein Wert von 0 anzeigt, dass keine Task die Motoren steuert. Wann immer eine Task etwas mit den Motoren tun möchte, führt sie die folgenden Befehle durch:

```
until (sem == 0);
sem = 1;
// die Motoren führen gerade Befehle aus
sem = 0;
```

Damit warten wir zuerst, bis keine andere Task die Motoren mehr benötigt (sem = 0). Dann übernehmen wir die Steuerung, indem wir sem = 1 setzen. Damit signalisieren wir den anderen Tasks, dass wir die Motoren gerade steuern. Wenn wir damit fertig sind, stellen wir sem zurück auf 0. Daran erkennen die anderen Tasks, dass wir die Motoren nicht mehr benötigen. Hier ist das Programm von oben, dieses Mal mit einem Semaphor. Wenn der Berührungs- Sensor etwas berührt, ist das Semaphor gesetzt und das Ausweichmanöver wird durchgeführt. Während dieser Prozedur läuft, muß die Task `move_square()` warten. Im dem Moment, wo das Ausweichmanöver beendet ist, wird das Semaphor = 0 gesetzt und `move_square()` kann fortfahren.

```
int sem;

task main()
{
  sem = 0;                          // Ausgangswert für die Variable
  start move_square;
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  while (true)
  {
    if (SENSOR_1 == 1)              // Signal von Berührungsensor
    {
      until (sem == 0); sem = 1; // Motor frei? Signalisiere “Motoren
belegt”
      OnRev(OUT_A+OUT_C); Wait(50); // Übernehme Steuerung!
      OnFwd(OUT_A); Wait(85);
      sem = 0;                      // signalisiere “Motoren frei”
    }
  }
}

task move_square()
{
  while (true)
  {
    until (sem == 0); sem = 1; // Motor frei? Signalisiere “Motoren belegt”
    OnFwd(OUT_A+OUT_C);        // Übernehme Steuerung!
    sem = 0;                   // signalisiere “Motoren frei”
    Wait(100);
    until (sem == 0); sem = 1; // Motor frei? Signalisiere “Motoren belegt”
    OnRev(OUT_C);              // Übernehme Steuerung!
    sem = 0;                   // signalisiere “Motoren frei”
    Wait(85);
  }
}
```

Du könntest argumentieren, dass es in `move_square()` nicht notwendig ist, das Semaphor auf 1 und zurück auf 0 zu setzen. Auf den ersten Blick hast du Recht. Dennoch ist dieses sinnvoll. Der Grund ist, dass sich der `OnFwd()` Befehl in Wirklichkeit aus zwei Befehlen zusammensetzt (siehe Kapitel 9). Dieser Befehlsablauf sollte nicht durch eine andere Task unterbrochen werden.
Semaphore sind sehr nützlich und, wenn du schwierigere Programme mit parallelen Tasks schreibst, wirst du sie fast immer benötigen. Dennoch gibt es einige wenige Fälle, in denen auch diese Methode scheitert. Finde sie heraus.

11.3. Zusammenfassung

In diesem Kapitel betrachteten wir einige der Probleme, die auftreten können, wenn parallele Tasks dieselben Motoren ansteuern. Achte darauf, weil unvorhergesehene Verhaltensweisen deines Roboters oft darauf zurückzuführen sind.
Wir sahen zwei unterschiedliche Möglichkeiten solche Probleme zu lösen.

- Die erste Lösung stoppt und startet Tasks wieder, und stellt so sicher, dass nur eine kritische Task läuft.
- Die zweite Lösung verwendet Semaphore, um die Ausführung von Tasks zu steuern. Dieses garantiert, dass immer nur eine Task die Motoren steuert.

12.Die Roboter unterhalten sich

Wenn du mehr als einen RCX besitzt, oder wenn du zusammen mit deinen Freunden mehrere RCX hast, dann ist das genau das richtige Kapitel für dich.
Die Roboter können untereinander durch den Infrarot- Kanal verbunden sein. Damit können mehrere Roboter zusammen arbeiten (oder gegeneinander kämpfen). Auch kannst du größere Roboter mit zwei RCX so aufbauen, dass Sie sechs Motoren und sechs Sensoren (oder wenn du die Tricks aus Kapitel 10.3 anwendest, noch mehr) haben können.

Die Kommunikation zwischen den Robotern funktioniert im Prinzip so: Ein Roboter kann den Befehl `SendMessage()` verwenden (send = senden, message = Nachricht) um einen Wert , der zwischen 0 und 255 liegen darf, über den Infrarot-Kanal zu senden. Alle weiteren Roboter empfangen diese Meldung und speichern sie. Das Programm in einem dieser Roboter kann den Wert der letzten Meldung, die mit `Message()` empfangen wurde, abfragen. Aufgrund dieses Wertes kann das Programm den Roboter bestimmte Tätigkeiten durchführen lassen.

12.1. Ordnungen

Wenn du zwei oder mehrere Roboter hast, ist meist einer der Anführer, wir nennen ihn “Master”. Die anderen Roboter, welche vom Anführer die Befehle erhalten, nennen wir Slave (engl. = Sklave). Der Anführer schickt den Sklaven Anweisungen und diese führen sie aus. Machmal sollen die Sklaven aber auch Nachrichten an den Anführer schicken, z.B. den Wert eines Sensorsignals. Wir müssen also zwei Programme schreiben, eins für den Anführer und eines für den (die) Sklaven.
Wir nehmen an, dass wir gerade einen Sklaven haben. Fangen wir mit einem einfachen Beispiel an. Hier kann der Sklave drei unterschiedliche Anweisungen durchführen: “fahre geradeaus”, “fahre rückwärts” und “halte an”. Sein Programm besteht aus einer einfachen Schleife. In dieser Schleife stellt es den Wert der aktuellen Meldung mit dem `ClearMessage()` Befehl auf 0 ein. Dann wartet es, bis die Meldung ungleich 0 wird. Entsprechend dem Wert der Meldung führt es eine der drei Anweisungen aus. Ist hier das Programm.

```
task main()                                    // SLAVE
{
  while (true)
  {
    ClearMessage();                             // Löcht alte Meldung
    until (Message() != 0);                     // wartet, mit Programmablauf
                                                 // bis eine Meldung eintrifft
                                                 // und wertet sie dann aus.
    if (Message() == 1) {OnFwd(OUT_A+OUT_C);} //Je nach übergebenem Wert
    if (Message() == 2) {OnRev(OUT_A+OUT_C);} //fährt der Roboter vor, zurück
    if (Message() == 3) {Off(OUT_A+OUT_C);}   //oder er hält an.
  }
}
```

Der Anführer hat sogar ein noch einfacheres Programm Er sendet einfach die Meldungen, entsprechend den Anweisungen und wartet dann etwas. Im unten gezeigten Programm befiehlt er dem Sklaven zunächst vorwärts zu fahren, dann nach zwei Sekunden rückwärts, und dann, wieder nach zwei Sekunden, anzuhalten..

```
task main()          // MASTER
{
  SendMessage(1); Wait(200);
  SendMessage(2); Wait(200);
  SendMessage(3);
}
```

Nachdem du diese beiden Programme geschrieben hast, muß du sie auf die Roboter übertragen. Jedes Programm muß auf einen der Roboter übertragen werden. Stelle daher sicher, dass während der Übertragung nur ein RCX eingeschaltet ist, da er sonst auch das Programm des anderen empfängt.
Wenn du beide Programme übertragen hast, schalte beide Roboter ein und starte die Programme: zuerst das im Sklaven und dann das im Meister.
Wenn du mehrere Sklaven hast, mußt du das Sklavenprogramm zunächst der Reihe nach auf jeden einzelnen der Sklaven übertragen (nicht gleichzeitig! Siehe dazu weiter unten). Nun werden alle Sklaven das Selbe tun.
Die Art, auf welche wir die Roboter sich untereinander verständigen lassen, nennt man ein **Protokoll**: Wir bestimmen folgendes: “1” bedeutet “fahre geradeaus”, “2” bedeutet “fahre rückwärts” und “3” bedeutet “stopp”. Es ist sehr wichtig, solche Protokolle sehr sorgfältig zu definieren, insbesondere dann wenn du viele solche Kommandos bestimmst.
Wenn es zum Beispiel mehrere Sklaven gibt, könnten wir ein Protokoll definieren, mit dem wir zwei Zahlen hintereinander, mit einer kleinen Pause dazwischen, senden: die erste Zahl ist der “Name” des Sklaven, der angesprochen werden soll. Die zweite ist die tatsächliche Anordnung. Alle Sklaven überprüfen zunächst die erste Zahl und stellen damit fest, ob ihnen die nachfolgende Anweisung gilt. Damit führt nur der Sklave die Anweisung aus, dem sie auch gilt.

Man könnte das mit der Schule vergleichen. Wenn der Lehrer dich anspricht, dann reagierst du auf seine Worte, wenn er dagegen einen anderen Namen nennt, dann tust du nichts.
Wir benötigen dafür natürlich für jeden Sklaven einen eigenen "Namen" in Form einer Zahl zwischen 0 und 255 und damit für jeden Sklaven ein etwas verändertes Programm, das sich beispielsweise durch eine Konstante am Beginn unterscheidet.

12.2. Wählen eines Anführers.

Wenn wir mehrere RCX miteinander kommunizieren lassen, benötigt jeder ein eigenes Programm (siehe oben). Aber wäre es nicht viel einfacher, wenn wir nur ein Programm zu allen Robotern downloaden könnten? Aber dann stellt sich die Frage: wer ist der Anführer? Die Antwort ist einfach: lassen wir doch die Roboter entscheiden. Sie sollen sich einen Anführer wählen, dem die anderen folgen. Nur, wie sollen wir das programmieren? Eigentlich ist das ganz einfach. Wir lassen jeden Roboter etwas warten und dann eine Meldung senden. Derjenige, der eine Meldung zuerst sendet, ist der Anführer. Das könnte zwar zu Problemen führen, wenn zwei Roboter im exakt gleichen Augenblick ihre Meldung senden, aber das ist sehr unwahrscheinlich. Ist hier das Programm:

```
task main()
{
  ClearMessage();
  Wait(200);              // damit hast du Zeit alle Roboter einzuschalten
  Wait(Random(400));      // warte zwischen 0 und 4 Sekunden
  if (Message() > 0)      // ein anderer Roboter war schneller
  {
    start slave;          //starte daher das Programm für den Skaven
  }
  else                    // andernfalls:
  {
    SendMessage(1);       // ich war schneller und bin nun der Anführer
    Wait(400);            // und warte, damit alle anderen bereit sind
    start master;         // dann starte ich das Programm für den Anführer
  }
}

task master()
{
  SendMessage(1); Wait(200);
  SendMessage(2); Wait(200);
  SendMessage(3);
}

task slave()
{
  while (true)
  {
    ClearMessage();
    until (Message() != 0);
    if (Message() == 1) {OnFwd(OUT_A+OUT_C);}
    if (Message() == 2) {OnRev(OUT_A+OUT_C);}
    if (Message() == 3) {Off(OUT_A+OUT_C);}
  }
}
```

Übertrage das Programm nacheinander auf jeden einzelnen Roboter, wieso, steht weiter unten. Starte dann die Roboter gleichzeitig und beobachte, was passiert (das kann bis zu 8 Sekunden dauern). Einer von ihnen sollte zum Anführer werden und die anderen sollten diese Befehle ausführen. Falls einmal keiner zum Anführer wird, starte die Programme einfach nochmals.

12.3. Vorsichtsmaßnahmen

Wenn du mit mehreren Robotern arbeitest, solltest du einige Dinge beachten

Achte darauf, dass beim Übertragen eines Programms nur ein RCX eingeschaltet ist!

Wenn du ein Programm zum Roboter überträgst (downloaden = herunterladen), ist das nicht so, dass der Computer etwas "sagt" und der RCX nur "zuhört", sondern der RCX und der Computer "unterhalten" sich. Der Roboter sagt dem Computer, ob er das Programm -oder auch Teile davon- richtig empfängt. Der Computer reagiert darauf, indem er neue Stücke sendet, oder indem er Teile, welche der RCX nicht – oder nicht richtig- verstanden hat, wiederholt.
Wenn nun zwei Roboter gleichzeitig eingeschaltet sind, versuchen beide dem Computer zu erklären, ob sie das Programm richtig empfangen. Das gibt ein Durcheinander, welches der Computer nicht versteht, er weiß ja nicht, dass es zwei Roboter gibt. Das ist so, als wenn mehrere Leute mit der gleichen Stimme gleichzeitig auf dich einreden. In dem Stimmengewirr würdest du wahrscheinlich auch durcheinader kommen. Der RCX (und der Computer) machen daher Fehler, und dein Programm im RCX ist wahrscheinlich nicht zu gebrauchen.

Überprüfe daher immer bevor du ein Programm überträgst, dass nur ein Roboter eingeschaltet ist!

Überlege dir ein Protokoll, das Übertragungsfehler erkennt!

Wenn zwei Roboter (oder ein Roboter und der Computer) gleichzeitig Informationen senden, können diese verloren gehen, und ein Roboter kann nicht gleichzeitig Meldungen senden und empfangen.
Dieses ist kein Problem, wenn nur ein Roboter Meldungen sendet, es also nur einen Meister gibt. Wenn du dagegen ein Programm möchtest, bei dem auch die Sklaven Meldungen senden -beispielsweise wenn einer gegen ein Hindernis stößt- dann kann es passieren, dass in diesem Moment ein anderer Sklave auch eine Meldung hat, oder der Meister sendet einen Befehl, der dann nicht verstanden wird.
Deshalb muß ein Kommunikationsprotokoll so definiert werden, dass es Kommunikationsprobleme erkennt und behebt!
Wenn der Meister einen Befehl sendet, sollte er vom Sklaven eine Antwort erhalten, die ihm mitteilt, dass der Befehl verstanden wurde. Antwortet der Sklave nicht umgehend auf diesen Befehl, so muß der Meister diesen wiederholen.
Ein Programm, das dieses tut, könnte beispielsweise so aussehen:

```
do
{
  SendMessage(1);          //sendet Nachricht (hier "1")
  ClearMessage();          //löscht den Nachrichtenspeicher
  Wait(10);                //wartet, während die Nachricht übertragen wird
}
while (Message() != 255); //und wartet dann auf die Bestätigung des
anderen
```

Hier wird 255 als Bestätigung für einen richtig verstandenen Befehl verwendet.

Hier kannst du vielleicht auch wieder die Semaphore einsetzen (siehe Kapitel 10.2). Wenn ein Roboter gerade eine Meldung absetzt, sollte ein weiterer Roboter mit seiner Meldung warten, bis der andere fertig ist.

12.4. Der RCX im Flüstermodus

Wenn du mehrere Roboter gleichzeitig in Betrieb hast, ist es ab und zu sinnvoll, wenn nur diejenigen einen Befehl empfangen, die nahe genug sind. Dazu kannst du deinen RCX in den "Flüstermodus" schalten. Dieses geschieht mit dem Befehl `SetTxPower(TX_POWER_LO)` den du dem Programm des Meisters hinzufügst. In diesem Fall wird nur ein schwaches IR Signal gesendet, und nur die RCX in unmittelbarer Nähe "hören" die Befehle. Dieses ist insbesondere sinnvoll, wenn du einen größeren Roboter aus zwei RCX aufbaust.
Mit `SetTxPower(TX_POWER_HI)`, schaltest du den Roboter wieder auf "laut". Damit werden die Befehle wieder auch von weiter entfernten RCX empfangen.

12.5. Zusammenfassung

In diesem Kapitel lernten wir einige grundlegenden Gesichtspunkte für die Kommunikation zwischen Robotern.
Zur Kommunikation verwenden wir Befehle um Meldungen zu senden, zu löschen und zu überprüfen.
Wir sahen wie wichtig es ist, ein Protokoll für die Kommunikation zu definieren. Solche Protokolle spielen in jeder möglichen Form der Kommunikation zwischen Computern eine entscheidende Rolle.
Das ist so, wie wenn wir Menschen uns unterhalten. Als Kinder lernen wir die Bedeutung von Worten, wie "links" oder "rechts", und wenn wir deren Bedeutung verwechseln, kann das schlimme Folgen haben.
Wir sahen auch, dass es eine Anzahl von Beschränkungen in der Kommunikation zwischen Robotern gibt. Diese machen es sogar noch wichtiger gute Protokolle zu definieren.

13.Noch mehr Befehle

NQC hat eine Anzahl weiterer Befehle. In diesem Kapitel behandeln wir drei Arten:

- Befehle für die Timer, das ist eine Art Stoppuhr,
- Befehle, welche die Anzeige des RCX steuern, und
- Befehle für den Datalog, einem Datenspeicher im RCX.

13.1. Timer

Der RCX hat vier eingebaute Timer. Diese Timer „ticken“ in 1/10 Sekunden- Sprüngen. Sie werden von 0 bis 3 nummeriert. du kannst den Wert eines Timers mit dem Befehl `ClearTimer()` zurücksetzen und den aktuellen Wert des Timers mit `Timer()` abfragen. Hier ist ein Beispiel für den Einsatz eines Timers.
Das folgende Programm läßt den Roboter 20 Sekunden lang zufällig vor- oder zurückfahren.

```
task main()
{
  ClearTimer(0);
  do
  {
    OnFwd(OUT_A+OUT_C);
    Wait(Random(100));
    OnRev(OUT_C);
    Wait(Random(100));
  }
  while (Timer(0)<200);
  Off(OUT_A+OUT_C);
}
```

Vielleicht erinnerst du dich dabei an das Programm in Kapitel 4.2, das genau die gleiche Aufgabe hatte. Wenn du beide Programme miteinander vergleichst, wirst du feststellen, dass das Programm mit dem Timer einfacher ist. Gerade als Ersatz für den `wait()` –Befehl sind Timer bestens geeignet. Man kann den Ablauf eines Programms steuern, indem man einen Timer zurücksetzt und dann wartet, bis er einen bestimmten Wert erreicht hat. Im Gegensatz zum `Wait()`- Befehl wird dabei der Programmablauf nicht unterbrochen. Daher kann der Roboter zum Beispiel auf die Information von den Sensoren reagieren, obwohl der Timer läuft. Das folgende einfache Programm ist ein Beispiel dafür. Es läßt den Roboter geradeaus fahren bis entweder 10 Sekunden vorüber sind, oder der Berührungssensor gegen etwas stößt.

```
task main()
{
  SetSensor(SENSOR_1,SENSOR_TOUCH);
  ClearTimer(3);                              //löscht Timer 3
  OnFwd(OUT_A+OUT_C);
  until ((SENSOR_1 == 1) || (Timer(3) >100)); //Bedingung erfüllt?
  Off(OUT_A+OUT_C);
}
```

Vergiß nicht, dass Timer in 1/10 Sekunden- Schritten arbeitet, während der `Wait()` –Befehl 1/100 Sekunden benutzt.

13.2. Das Display des RCX

Es ist möglich, das Display des RCX auf zwei verschiedene Arten zu steuern. Zunächst kannst du bestimmen, welche Information dargestellt werden sollen: Die interne Uhrzeit des RCX, der Status eines Sensors oder Motors.
Dieses kannst du ja auch mit der schwarzen Taste “View” am RCX einstellen.
Diese Einstellungen kannst du aber auch mit einem Programm verändern, indem du den Befehl `SelectDisplay()` verwendest. Das folgende Programm zeigt alle sieben Möglichkeiten, eine nach der anderen.

```
task main()
{
  SelectDisplay(DISPLAY_SENSOR_1); Wait(100);  // Eingang 1
  SelectDisplay(DISPLAY_SENSOR_2); Wait(100);  // Eingang 2
  SelectDisplay(DISPLAY_SENSOR_3); Wait(100);  // Eingang 3
  SelectDisplay(DISPLAY_OUT_A);    Wait(100);  // Ausgang A
  SelectDisplay(DISPLAY_OUT_B);    Wait(100);  // Ausgang B
  SelectDisplay(DISPLAY_OUT_C);    Wait(100);  // Ausgang C
  SelectDisplay(DISPLAY_WATCH);    Wait(100);  // Systemuhr
}
```

Beachte,dass du nicht versehentlich `SelectDisplay(SENSOR_1)` schreibst.

Du kannst die Display- Anzeige aber auch steuern, indem du den den Wert des Systemtaktgebers veränderst.
Damit kannst du dir beispielsweise Diagnose- Informationen anzeigen lassen. Wenn du diesen Befehl mit unterschiedlichen Werten an verschiedenen Stellen in deinem Programm einbaust, kannst du am Display immer erkennen, an welcher Stelle im Programm sich der RCX gerade befindet.
Verwende dafür den Befehl `SetWatch()`. Hier ist ein kleines Programm dazu:

```
task main()
{
  SetWatch(1,1); Wait(100);
  SetWatch(2,4); Wait(100);
  SetWatch(3,9); Wait(100);
  SetWatch(4,16); Wait(100);
  SetWatch(5,25); Wait(100);
}
```

Beachte, dass in `SetWatch()` nur Konstanten zulässig sind.

13.3. Speichern von Informationen

Der RCX kann Werte von Variablen, Sensoren und Timern speichern. Dies geschieht in einem Speicherbereích des RCX, der “datalog” genannt wird. Die Werte im datalog können leider nicht innerhalb des RCX verwendet werden, aber sie können durch deinen Computer gelesen werden. Dieses hilft dir, wenn du überprüfen möchtest, was in deinem Roboter vorgeht. Das **RCX Command Center** hat ein spezielles Fenster, in dem du den aktuellen Inhalt des datalog ansehen kannst.

Um den Datalog zu verwenden, benötigst du drei Schritte:

- Zuerst muß das NQC-Programm die Größe des datalog mit dem Befehl `CreateDatalog()` definieren. Dieses löscht auch den aktuellen Inhalt vom datalog.
- Als nächstes können Werte in den datalog mit dem Befehl `AddToDatalog()` geschrieben werden. Die Werte werden einer nach dem anderen geschrieben. Wenn du das Display des RCX betrachtest, siehst du wie rechts neben der Programmnummer nacheinander die vier Viertel eines Kreises erscheinen. Wenn der Kreis komplett ist, ist der datalog voll. Wenn das Ende des datalog erreicht wird, geschieht nichts mehr. Neue Werte werden nicht mehr gespeichert.
- Im dritte Schritt laden wir den datalog in den PC. Wähle dafür im **RCX-Command Center** den Befehl Datalog im Menue Tools (engl. = Werkzeuge, Hilfsmittel). Betätige dort die Taste “upload” um die Werte aus dem Datalog des RCX anzuzeigen.
  Den datalog kannst du aber auch für andere Zwecke verwenden. Findige Lego-Mindstorms-Programmierer haben ihn schon verwendet um damit eine Scanner zu bauen.
  Hier ist ein einfaches Beispiel, mit dem du die Veränderung der Helligkeit in deinem Zimmer erfassen kannst. Das Progamm läuft 20 Sekunden. Die Werte des Lichtsensors werden fünfmal pro Sekunde in den datalog geschrieben. Damit kannst du beispielsweise das Einschaltverhalten einer Energiesparlampe mit dem einer Glühbirne vergleichen.

```
task main()
{
  SetSensor(SENSOR_2,SENSOR_LIGHT);
  OnFwd(OUT_A+OUT_C);
  CreateDatalog(50);
  repeat (50)
  {
    AddToDatalog(SENSOR_2);
    Wait(20);
  }
  Off(OUT_A+OUT_C);
}
```

14.NQC- Befehle im Überblick

Hier siehst du eine Liste aller Statements, Bedingungen, Befehle und Ausdrücke des RCX. Die meisten wurden in den Kapiteln 1 bis 12 behandelt. Daher werden sie hier nur kurz beschrieben.
Da das alles etwas schwierig zu verstehen ist, habe ich dir jeweils ein paar Beispiele für den RCX erstellt. Schreibe das Programm ab, übertrage es auf den RCX und laß' es dann laufen. Übertrage anschließend den Datalog des RCX mit Hilfe des **RCX-Command Centers** auf deinen Computer und vergleiche die Ergebnisse. Verändere dann das Programm, rechne das Ergebnis aus, übertrage das Programm und überprüfe das Ergebnis des Datalog mit dem von dir berechneten. Wenn du das ein paarmal richtig gemacht hast, dann hast du die Funktion verstanden.
Für weitere Informationen schaue bitte im *NQC_Guide.doc* nach, das allerdings in enschlischer Sprache verfaßt ist.

Statements

Statements sind Schlüsselworte, die ein Programm anweisen etwas zu auszuführen.

Statement	**Beschreibung**
while (*Bedingung*) *Anweisung*	Führe *Anweisung* nie oder solange aus, wie die *Bedingung* zutrifft
do *Anweisung* **while** (*Bedingung*)	Führe *Anweisung* einmal oder solange aus, wie die *Bedingung* zutrifft
until (*Bedingung*) *Anweisung*	Führe *Anweisung* nie oder solange aus, bis die *Bedingung* zutrifft
break	Unterbreche die Ausführung der **while**/**do**/**until** Anweisung
continue	Springe zum nächsten Durchlauf der **while**/**do**/**until** Anweisung
repeat (*Wert*) *Anweisung*	Wiederhole die *Anweisung* entsprechend dem **Wert** in Klammern
if (*Bedingung*) *Anweisung 1*	Führe *Anweisung 1* aus, wenn die *Bedingung* zutrifft
if (*Bedingung*) *Anweisung 1* **else** *Anweisung 2*	Führe *Anweisung 1* aus, wenn die *Bedingung* zutrifft, andernfalls führe *Anweisung 2* aus
start *task_name*	Starte die genannte Task
stop *task_name*	Stoppe die genannte Task
Funktion(*Argumente*)	Rufe die Funktion auf und übergebe dabei die Argumente (Zahlen)
Variable = Formel	Berechne *Formel* und weise den Wert *Variable* zu
Variable += Formel	Berechne *Formel* und addiere den Wert zu *Variable* hinzu
Variable -= Formel	Berechne *Formel* und subtrahiere den Wert von *Variable*
*Variable *= Formel*	Berechne *Formel*, multipliziere den Wert mit *Var* und weise ihn *Var* zu
Var /= Formel	Berechne *Formel*, dividiere *Var* durch den Wert und weise ihn *Var* zu
Variable |= Formel	Berechne *Formel* und erhöhe die Variable um ein Bit, wenn das Bitmuster der Variablen von rechts mit dem der Formel übereinstimmt [1)]
Variable &= Formel	Berechne *Formel* und erhöhe die Variable um ein Bit, wenn das Bitmuster der Variablen von links mit dem der Formel übereinstimmt [1)]
return	Kehre zur aufrufenden Stelle zurück (Dient dazu Unterprogramme vorzeitig zu verlassen, wenn z.B. eine if-Bedingung erfüllt ist)
expression	Einfach nur ein Ausdruck, wie x=5; oder nur ein ;

[1)] kann ich leider nicht besser erklären. Vielleicht hilft dir das Beispiel auf der nächsten Seite weiter oder es findet sich ein Mathematiker, der mir da helfen kann.

Beispielprogramm: (hier das Programm aus Kapitel 4)

```
int aa;               // Definition der Variablen aa
int bb, cc;           // gleichzeitige Definition der Variablen bb und cc
task main()
{
  CreateDatalog (10); //reserviere einen Speicherbereich von 10 Plätzen im Datalog
  aa = 10;            // weise der Variablen aa den Wert 10 zu
  bb = 20 * 5;        // weise bb den Wert 20 mal 5 = 100 zu
  cc = bb;            // sezte cc gleich bb, cc wird also ebenfalls 100
  cc /= aa;           // berechne cc / aa und weise das Ergenis cc zu, cc = 100/10 = 10
  cc -= 1;            // ziehe von cc 1 ab und weise das Ergebnis cc zu, cc wird also 9
  aa = 10 * (cc + 3); // berechne 10 * (cc + 3), aa wird also 10 * (9 + 3) = 120
  AddToDatalog (aa);  //füge aa dem Datalog hinzu
  AddToDatalog (bb);  //füge bb dem Datalog hinzu
  AddToDatalog (cc);  //füge cc dem Datalog hinzu
}
```

Beispiel für ***Variable*** **|= *Formel***:

```
int aa, bb;            // gleichzeitige Definition der Variablen aa und bb
task main()
{
  CreateDatalog (200); //reserviere einen Speicherbereich von 100 Plätzen im Datalog
  aa = 0;              // aa = 0
  bb = 0;
  repeat (100)
  {
  aa = aa + 1;
  bb |= aa;           // eine Funktion, deren Aufgabe/ Sinn ich nicht beschreiben kann
  AddToDatalog (aa);  //füge aa dem Datalog hinzu
  AddToDatalog (bb);  //füge bb dem Datalog hinzu
  }
}
```

Beispiel für ***Variable*** **&= *Formel***:

```
int aa, bb;            // gleichzeitige Definition der Variablen aa und bb
task main()
{
  CreateDatalog (200); //reserviere einen Speicherbereich von 100 Plätzen im Datalog
  aa = 0;              // aa = 0
  repeat (100)         //wiederhole folgendes 100 mal
  {
  aa = aa + 1;
  bb = 8;
  bb &= aa;           // eine Funktion, deren Aufgabe/ Sinn ich nicht beschreiben kann
  AddToDatalog (aa);  //füge aa dem Datalog hinzu
  AddToDatalog (bb);  //füge bb dem Datalog hinzu
  }
}
```

Bedingungen

Bedingungen werden dazu verwendet Programmabläufe, abhängig vom Wahrheitsgehalt der Bedingung, zu steuern. Meistens beinhaltet eine Bedingung den Vergleich zwischen Ausdrücken.

Bedingung	**Bedeutung**
`true`	Immer **wahr**
`false`	Immer **falsch**
`Ausdruck 1 == Ausdruck 2`	Prüfe, ob die *Ausdrücke* gleich sind
`Ausdruck 1 != Ausdruck 2`	Prüfe, ob die *Ausdrücke* nicht gleich sind
`Ausdruck 1 < Ausdruck 2`	Prüfe ob *Ausdruck 1* kleiner als *Ausdruck 2* ist
`Ausdruck 1 <= Ausdruck 2`	Prüfe ob *Ausdruck 1* kleiner oder gleich *Ausdruck 2* ist
`Ausdruck 1 > Ausdruck 2`	Prüfe ob *Ausdruck 1* größer als *Ausdruck 2* ist
`Ausdruck 1 >= Ausdruck 2`	Prüfe ob *Ausdruck 1* größer oder gleich *Ausdruck 2* ist
`! Bedingung`	Logische Umkehrung der Bedingung: was **wahr** ist, wird **falsch**, was **falsch** ist, wird **wahr**
`Bedingung 1 && Bedingung 2`	Logisches UND der beiden *Bedingungen* Nur dann **wahr**, wenn beide *Bedingungen* **wahr** sind
`Bedingung 1 || Bedingung 2`	Logisches ODER der beiden Bedingungen **Wahr**, wenn eine der beiden Bedingungen **wahr** ist

```
int aa, bb;       // gleichzeitige Definition der Variablen aa und bb
int wahr, falsch; // die Variablen wahr und falsch (Variablen 2 und 3 im Datatlog
task main()
 {
  aa = 5; bb = 5;
  CreateDatalog (10); //reserviere einen Speicherbereich von 100 Plätzen im Datalog
  wahr = 0 ;falsch = 0;                    // setzt Variablen zurück
  if (true) wahr = 1; else falsch =1;      /* "true" ist immer wahr,
                                              also ist wahr = 1 und falsch = 0 */
  AddToDatalog (wahr);
  AddToDatalog (falsch);
  wahr = 0; falsch = 0;
  if (aa == bb) wahr = 1; else falsch =1;   /* aa und bb sind beide 5 und damit
gleich,
                                               also ist wahr = 1 und falsch = 0 */
  AddToDatalog (wahr);
  AddToDatalog (falsch);
  wahr = 0; falsch = 0;
  if (aa != bb) wahr = 1; else falsch =1;  /* aa und bb sind beide 5, und damit nicht
                                              ungleich, also ist wahr = 0
                                              und falsch = 1 */
  AddToDatalog (wahr);
  AddToDatalog (falsch);
  }
```

Ausdrücke

Es gibt eine Anzahl verschiedener Werte, die mit Ausdrücken (mathematische Formeln), Konstanten, Variablen und Sensor-Signalen verknüpft werden können. Beachte, dass SENSOR_1, SENSOR_2, und SENSOR_3 Makrofunktionen sind, die auf SensorValue(0), SensorValue(1), und SensorValue(2) zurückgreifen.

Wert	**Beschreibung**
Zahl	Ein konstanter Wert (z.B.: “123”)
Variable	Ein Name, der stellvertretend für irgendeine Zahl steht (z.B.: “x”)
Timer(*n*)	Der Wert von Timer n, wobei n zwischen 0 und 3 liegt
Random(*n*)	Eine Zufallszahl zwischen 0 und n
SensorValue(*n*)	Der derzeitige Wert, den der Sensor n liefert, wobei n 0, 1 oder 2 ist.
Watch()	Der derzeitige Wert der Systemuhr (= Systemuhrzeit)
Message()	Wert der zuletzt von der Infrarot- Schnittstelle empfangenen Nachricht.

```
int aa;
task main()
 {
  aa = 77;
  SetWatch (14,33);                  /* Setzt die Systemuhr auf 14 Stunden und
                                        33 Minuten (= 873 Minuten) */
  SetSensor(SENSOR_1,SENSOR_LIGHT); // Setzt SENSOR_1 als Lichtsensor
  CreateDatalog (10);                /* reserviert einen Speicherbereich von 10 Plätzen
                                        im Datalog */
  AddToDatalog (123);                // Überträgt den Wert 123 in den Datalog
  AddToDatalog (aa);                 // Übertägt den Wert von aa (=77) in den Datalog
  AddToDatalog (Timer(0));           /* Überträgt den momentanen Wert des
                                        Systemtimers 0 in den Datalog */
  AddToDatalog (Random (100));       // Übertägt den Wert des Displays in den Datalog
  AddToDatalog (SensorValue(0));     /* Überträgt den Wert des Lichtsensors in den
                                        Datalog (im Makro SENSOR_1, RCX-intern aber
                                        Sensor (0) !! ) */
  AddToDatalog (Watch());            /* Überträgt die Anzeige der Systemuhr in Minuten
                                        in den Datalog */
  AddToDatalog (Message());          /* Übertägt den Wert der zuletzt empfangenen
                                        Nachricht in den Datalog
                                        Wenn kein anderer RCX eine Nachricht gesendet
                                        hat, ist der Wert 0*/
  SetSensor(SENSOR_1,SENSOR_TOUCH); /* Setzt SENSOR_1 als Berührungssensor,
                                        damit die LED aus geht. */
 }
```

Mathematische Operationen

Werte können durch mathematische Operationen verknüpft werden. Viele der Operationen können nur in konstanten Ausdrücken verwendet werden. Das bedeutet, dass sie entweder selbst Konstanten sein müssen, oder der Ausdruck nur Konstanten enthalten darf.
Sorry, leider habe ich auch hier nicht alle Eigenheiten durchschaut.

Operator	Beschreibung	Bezug	Einschränkung	Beispiel
`abs()`	Absoluter Wert	n/a		`abs(-3) wird 3`
`sign()`	Vorzeichen	n/a		`sign(-3) gibt -1`
`++`	Addition	left	Nur Variablen	`x++ or ++x`
`--`	Subtraktion	left	Nur Variablen	`x-- or --x`
`-`	Vorzeichenwechsel	rechts	Nur Konstanten	`-x`
`~`	Bitwise negation (unary)	rechts		`~123`
`*`	Multiplikation	links		`x * y`
`/`	Division	links		`x / y`
`%`	Modulo (ganzzahliger Rest)	links		`123 % 4`
`+`	Addition	links		`x + y`
`-`	Subtraktion	links		`x - y`
`<<`	Left shift	links	Nur Konstanten	`123 << 4`
`>>`	Right shift	links	Nur Konstanten	`123 >> 4`
`&`	Bitwise UND	links		`x & y`
`^`	Bitweises XOR	links	Nur Konstanten	`123 ^ 4`
`|`	Bitweises ODER	links		`x | y`
`&&`	Logisches UND	links	Nur Konstanten	`123 && 4`
`||`	Logisches ODER	links	Nur Konstanten	`123 || 4`

RCX Funktionen

Bei den meisten Funktionen werden als Argumente konstante Ausdrücke (eine Zahl oder andere konstante Ausdrücke) benötigt. Ausnahme sind Funktionen, die einen Sensor als Argument verwenden, und solche, bei denen man alle Ausdrücke verwenden kann. Im Falle eines Sensors ist das Argument ein Sensorname: SENSOR_1, SENSOR_2, or SENSOR_3. In manchen Fällen sind es die zuvor definierte Namen (z.B. SENSOR_TOUCH) für passende Konstanten.

Funktion	**Beschreibung**	**Beispiel**
`SetSensor(Sensor, Einstellung)`	Richtet einen Sensor ein	`SetSensor(SENSOR_1, SENSOR_TOUCH)`
`SetSensorMode(Sensor, Art)`	Richtet die Art ein, wie Signale erfaßt werden	`SetSensor(SENSOR_2, SENSOR_MODE_PERCENT)` `SetSensor(SENSOR_2, SENSOR_MODE_RAW)`
`SetSensorType(Sensor, Typ)`	Bestimmt den Sensortyp	`SetSensor(SENSOR_2, SENSOR_TYPE_LIGHT)`
`ClearSensor(Sensor)`	Löscht den Meßwert	`ClearSensor(SENSOR_3)`
`On(Ausgänge)`	Schaltet einen oder mehrere Ausgänge ein	`On(OUT_C)` `On(OUT_A+OUT_B)`
`Off(Ausgänge)`	Schaltet einen oder mehrere Ausgänge aus	`Off(OUT_C)`
`Float(Ausgänge)`	Läßt Ausgang auslaufen	`Float(OUT_B)`
`Fwd(Ausgänge)`	Schaltet Vorwärtsgang ein	`Fwd(OUT_A)`
`Rev(Ausgänge)`	Schaltet Rückwärtsgang ein	`Rev(OUT_B)`
`Toggle(Ausgänge)`	Schaltet Drehrichtung um	`Toggle(OUT_C)`
`OnFwd(Ausgänge)`	Schaltet Motor in Vorwärtsrichtung ein	`OnFwd(OUT_A)`
`OnRev(Ausgänge)`	Schaltet Motor in Rückwärtsrichtung ein	`OnRev(OUT_B)`
`OnFor(Ausgänge, Zeit)`	Schaltet den Motor für eine gewisse Zeit (1/100s) ein. Zeit kann eine Variable sein.	`OnFor(OUT_A, 200)` `OnFor(OUT_A+OUT_C, tt)`
`SetOutput(Ausgänge, Art)`	Schaltet Ausgang ein oder aus	`SetOutput(OUT_A, OUT_ON)`
`SetDirection(Ausgänge, dir)`	Stellt Drehrichtung ein	`SetDirection(OUT_A, OUT_FWD)` `SetDirection(OUT_C, OUT_REV)`

`SetPower(Ausgänge, Leistung)`	Stellt die Leistung der Motoren ein (0-7). „Leistung“ kann eine Variable sein	`SetPower(OUT_A, 6)`
		`SetPower(OUT_A+OUT_C, 3)`
		`SetPower(OUT_B, Power)`

`Wait(Zeit)`	Unterbricht den Programmablauf für „Zeit“. Zeit ist in 1/100 s und kann eine Variable sein	`Wait(x)`
`PlaySound(sound)`	Spielt den genannten sound sound (0-5).	`PlaySound(SOUND_CLICK)` `PlaySound(3)`
`PlayTone(freq, dauer)`	Spielt einen Ton mit der genannten Frequenz und Dauer (in 1/10 s)	`PlayTone(440, 5)`
`ClearTimer(timer)`	Stellt den Timer (0 – 3) auf Null zurück	`ClearTimer(0)`
`StopAllTasks()`	Stoppt alles derzeit laufenden Tasks	`StopAllTasks()`
`SelectDisplay(mode)`	Wählt eine der 7 möglichen Anzeigemodi aus.: 0: Systemuhr, 1-3: Sensoren, 4-6: Einstellung der Ausgänge. Mode kann eine Variable sein.	`SelectDisplay(1)` `SelectDisplay(mode)`
`SendMessage(message)`	Sende eine IR-Nachricht (1-255). Message kann eine Variable sein	`SendMessage(x)` `SendMessage(255)`

`ClearMessage()`	Löscht Nachrichtenspeicher	`ClearMessage()`
`CreateDatalog(Größe)`	Erstelle ein neues datalog mit der genannten Größe	`CreateDatalog(100)`
`AddToDatalog(Wert)`	Füge dem datalog einen Wert hinzu.	`AddToDatalog(Timer(0))`
`SetWatch(Stunden, Minuten)`	Stellt die Systemuhr	`SetWatch(1,30)`
`SetTxPower(hi_lo)`	Stellt den Infarrotsender auf kurze oder große Reichweite	`SetTxPower(TX_POWER_LO)`

RCX Konstanten

Viele der RCX- Funktionen haben Namen erhalten, die das Programm lesbarer gestalten helfen.

Sensoreinstellung für SetSensor()	`SENSOR_TOUCH, SENSOR_LIGHT, SENSOR_ROTATION, SENSOR_CELSIUS, SENSOR_FAHRENHEIT, SENSOR_PULSE, SENSOR_EDGE`
Sensormodi SetSensorMode()	`SENSOR_MODE_RAW, SENSOR_MODE_BOOL, SENSOR_MODE_EDGE, SENSOR_MODE_PULSE, SENSOR_MODE_PERCENT, SENSOR_MODE_CELSIUS, SENSOR_MODE_FAHRENHEIT, SENSOR_MODE_ROTATION`
Sensortypen SetSensorType()	`SENSOR_TYPE_TOUCH, SENSOR_TYPE_TEMPERATURE, SENSOR_TYPE_LIGHT, SENSOR_TYPE_ROTATION`
Ausgänge On(), Off(), etc.	`OUT_A, OUT_B, OUT_C`
Einstellungen der Ausgänge SetOutput()	`OUT_ON, OUT_OFF, OUT_FLOAT`
Drehrichtung SetDirection()	`OUT_FWD, OUT_REV, OUT_TOGGLE`
Leistung SetPower()	`OUT_LOW, OUT_HALF, OUT_FULL`
Sounds PlaySound()	`SOUND_CLICK, SOUND_DOUBLE_BEEP, SOUND_DOWN, SOUND_UP, SOUND_LOW_BEEP, SOUND_FAST_UP`
Einstellungen SelectDisplay()	`DISPLAY_WATCH, DISPLAY_SENSOR_1, DISPLAY_SENSOR_2, DISPLAY_SENSOR_3, DISPLAY_OUT_A, DISPLAY_OUT_B, DISPLAY_OUT_C`
IR-Reichweite SetTxPower()	`TX_POWER_LO, TX_POWER_HI`

Schlüsselworte

Schlüsselworte sind solche Worte, die vom NQC- Compiler (Übersetzungsprogramm) benötigt werden. Es ist nicht zulässig sie in irgendeiner anderen Weise zu verwenden.

Das gilt für folgende Wörter: **`__sensor, abs, asm, break, const, continue, do, else, false, if, inline, int, repeat, return, sign, start, stop, sub, task, true, void, while.`**

15. Abschließende Bemerkungen

Wenn du dich durch dieses Benutzerhandbuch gearbeitet hast, kannst du dich als Experten in NQC bezeichnen. Falls nicht, ist es an der Zeit damit anzufangen.
Damit und mit Kreativität kannst du die tollsten Lego- Roboter bauen und die erstaunlichsten Dinge tun.
Diese Anleitung umfaßte nicht alle Aspekte des **RCX-Command Center**. Auch NQC ist noch in der Entwicklung. Zukünftige Versionen können zusätzliche Funktionen enthalten.
In dieser Anleitung wurden viele Möglichkeiten, die du mit NQC hast, nicht behandelt. Für weitere Informationen schaue bitte im **NQC-Guide** nach. Das ist ein Word-Dokument, das beim RCX_CC im Pfad documents zu finden ist.

Im folgenden sind hier noch ein paar Internet- Links (Link = Verbindung zu einer anderen Internet-Adresse) genannt. Du findest dort viele Anregungen und Hilfestellungen. Allerdings sind die meisten Links in Englisch. Du kannst dir aber diese Seiten online übersetzten lassen, beispielsweise mit einem Übersetzungsprogramm:

http://www.heisoft.de

Gib dort die Internetadresse der Seite an, die du übersetzt haben möchtest. Du bekommst dann die Seite (in einem manchmal fürchterlichen deutsch) angezeigt. Nimm dazu am besten die Adresse mit Strg+C in die Zwischenablage deines Computers und füge sie mit Strg+V beim Übersetzungsprogramm ein.
Ansonsten hilft dir nur dein Englisch- Unterricht in der Schule. Aber vielleicht hilft dir Lego-Mindstorms auf diese Weise bessere Noten in Englisch zu bekommen.

http://www.legomindstorms.com/

Das WorldWideWeb ist eine tolle Quelle für weitere Information. Probiere einfach ein paar Links auf der Homepage von Mark Overmars

http://www.cs.uu.nl/people/markov/lego/

und LUGNET, the LEGO® Users Group Network (unofficial):

http://www.lugnet.com/

Viele Informationen kannst du auch in der lugnet.robotics Newsgroup bekommen.

Anmerkungen des Übersetzers.

Auch ich habe das Online- Übersetzungs- Programm verwendet, um eine rohe Übersetzung für diese Anleitung zu bekommen. Danach aber war für mich viel Handarbeit angesagt. Während der Übersetzungsarbeit habe ich viele Programme ausprobiert und erweitert. Dabei habe ich auch eine Menge über NQC gelernt.
Leider habe ich nicht alle Redewendungen und Kniffe durchschaut. Insbesondere das Kapitel 13 (NQC- Befehle im Überblick) hat mir große Schwierigkeiten bereitet, da ich hier den Sinn einiger der dargestellten Funktion nicht durchschaut habe. Vieles habe ich dabei durch Ausprobieren herausgefunden. Daher rate ich auch dir: probiere einfach alles Mögliche aus.
Lego-Mindstorms macht viel Spaß. Mit **NQC** von Dave Baum und dem **NQC Command Center** von Marc Overmars bekommen wir ein tolles Werkzeug, mit dem unsere Lego- Roboter noch leistungsfähiger werden.

Danke Dave !
Danke Mark !

Ich hoffe, dass ich bei der Übersetzung keine gravierenden Fehler gemacht habe.

Bad Mergentheim, im Januar 2000.

Martin Breuner